1986-12

A prototype visual structure editor for Pascal

Farley, Michael F.

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A PROTOTYPE VISUAL STRUCTURE EDITOR
FOR PASCAL

by

Michael F. Farley

December 1986

Thesis Advisor: Daniel L. Davis

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (If applicable) Code 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING/SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
A PROTOTYPE VISUAL STRUCTURE EDITOR FOR PASCAL

12 PERSONAL AUTHOR(S)
Farley, Michael F.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM ________ TO ________	1986 December	78

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Visual programming, structure editor, Pascal, programming environment, interactive interface

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The development of programming tools for conventional, textual environments has dramatically increased the productivity of the individual programmer, but these environments have been developed to their logical extremes. Current research in the field of interactive programming environments has moved toward graphics-oriented systems to take advantage of the wider bandwidth of information transfer that is inherent in these systems. This paper describes the design and implementation of a prototype visual programming paradigm. Built around an interactive, user-friendly interface which uses a mouse, menus and windows, the system enables the user to construct Pascal programs through a combination of graphical object manipulations and textual entries.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL Daniel L. Davis	22b TELEPHONE (Include Area Code) 408-646-3091	22c OFFICE SYMBOL 52Dv

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

Unclassified

Approved for public release; distribution is unlimited.

A Prototype Visual Structure Editor for Pascal

by

Michael F. Farley
Lieutenant Commander, United States Navy
B.S., The University of Louisville, 1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1986

ABSTRACT

The development of programming tools for conventional, textual environments has dramatically increased the productivity of the individual programmer, but these environments have been developed to their logical extremes. Current research in the field of interactive programming environments has moved toward graphics-oriented systems to take advantage of the wider bandwidth of information transfer that is inherent in these systems. This paper describes the design and implementation of a prototype visual programming paradigm. Built around an interactive, user-friendly interface which uses a mouse, menus and windows, the system enables the user to construct Pascal programs through a combination of graphical object manipulations and textual entries.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. THE PROGRAM DEVELOPMENT PROBLEM

Engineering software can be a formidable task. This notion is based on the limited capacity of the unaided human intellect to fully comprehend and track all aspects of even a moderately large computer program. These difficulties are further exacerbated by an commensurate increase in the size and complexity of the problems to be solved using computers. Successful software design requires that careful consideration be given to many issues which may affect the quality of the final product. They include: correctness of program specifications, life-cycle expectancy, projected maintenance requirements and development methodology. Additionally, large projects normally require coordinating the efforts of a multitude of programmers, each with different tasks to accomplish in support of that project. Although each of these issues is important, we shall focus our attention on the specific problem of increasing individual programmer productivity. Programming aids such as high-level languages and tools such as full-screen structure editors and syntax-directed text editors have helped in this area. The increase in productivity realized by the use of these devices has been significant, allowing the creation of programs with fewer keystrokes in less time and generally providing programmers a better grasp on the management of program development. The evolution of programming tools has also helped to reduce the time spent in the programming cycle.

1. The Programming Cycle

The majority of programs are created using a conventional development scheme called the programming cycle. This scheme involves several phases: Edit, Compile, Link, Execute and Debug. Errors will generally occur in the course of developing a program. The source of these errors may be the result of improper design logic, syntactic oversights, typographical errors, or semantic errors. These errors can manifest themselves during any phase, requiring another iteration of the cycle (except, of course, for errors detected during the Edit phase). The conventional interpreted system shortens this cycle somewhat by eliminating the compilation and link steps, but this happens at the expense of speed and efficiency of program execution. The conventional interactive system provides still greater flexibility through the use of its tools and other programming aids. Instead of sequentially stepping through the Edit-Run-Debug cycle as in the interpretive system, the user can easily transition directly from any state to any other state. An excellent example of such a system is MacPascal, an interactive, textual interpreted system which runs on the Apple Macintosh. The system provides syntactic editing, allows the user to step through a program while viewing the results of program execution, and provides a facility to view the results of performing an immediate execution of a disconnected code segment.

2. A Possible Solution

What then should be our choice of a system with which to do program development? One solution to this problem is to fully develop

the program to its final form on an interactive, interpreted system, and then port the polished source code to an efficient compiler to produce a quality product. This is a simplistic view which doesn't take into account the many peripheral problems of program development such as portability and compatibility. However, if the systems used adhere to some sort of language standard, this solution may be made viable by ensuring that the intermediate product, i.e., the source code, is in that standard form. There are already a great number of excellent compilers available to handle source code written in the currently popular languages. If we wish to increase programmer productivity, our goal must be to design an effective interactive programming environment which is built around a standardized high-level language. Textual environments have been exploited to their logical extreme, so it seems that a practical alternative is to explore the utilization of graphical techniques.

B. THE VISUAL APPROACH

Current research in the field of interactive programming environments has moved toward graphics-oriented or "visual" systems to take advantage of the wider bandwidth of information transfer that is inherent in these systems. The guiding principle behind this movement is improvement of user-system communication. There are three ways to do this: improve the interface channel capacity, improve the sophistication of the system's ability to process information, or improve the sophistication of the user's ability to process information. An example of the last is the UNIX interface's attempts to optimize a poor channel through the use of short

cryptic messages, both to and from the user. This is contrary to our desire to ease the user's burden. The visual approach concentrates on the first and second options, providing facilities to enable a system to be used and understood by even casual users. The latest development in visual systems is the user-friendly interface, which makes use of menus, a pointing device and integrated graphics to improve user-system interactions.

C. DESIRABLE PROPERTIES OF A VISUAL INTERFACE

The properties that contribute to the design of an effective interactive interface have been well documented in the literature. Hansen presented his "User Engineering Principles", which were used in the design of the Emily text editing system [Ref. 1]. Two of these stand out as significantly relevant to visual interfaces: Minimizing Memorization and Engineering For Errors. The former refers to the features of a system which aid the user by displaying a list of descriptive choices to the user rather than making him remember commands or file names, while the latter means understanding that users will make errors, so common errors are anticipated and either prevented or minimized and error messages are made understandable rather than cryptic.

The User Interface Guidelines for the Apple Macintosh specify three qualities as necessary for allowing a user to feel in control of the computer. They are responsiveness, permissiveness and, most importantly, consistency. Responsiveness means that the user's actions tend to have direct results, and he is able to accomplish what needs to be

done spontaneously and intuitively without having to set up a chain of sequential events or commands. Permissiveness means that the user is allowed to do any reasonable action, is not subjected to an overabundance of error messages, and is not forced to constantly operate in modes. Consistency means that the interface operations remain consistent from application to application. Menu operations, file operations and edit operations are all accomplished in the same manner, regardless of the application program's purpose. [Ref. 2]

D. SCOPE

Realistically, it is unfeasible to expect a fully functional programming environment to be the product of this research. Keeping in mind the properties we have discussed, we shall concentrate our efforts on designing a prototype, visual programming paradigm with a user-friendly interface in such a manner as to be extendable to incorporate an interpreter, debugger and other useful tools. Portability requirements and a desire to incorporate the discipline of Structured Programming into the environment dictates selection of a high-level language that conforms to the imperative, procedural paradigm.

II. DESIGN ISSUES

The selection of a user-friendly interface for our prototype presents us with an abundance of design choices regarding its "Look and Feel." The only true measure of the success of our design is the ease with which a user can efficiently and effectively transfer his thoughts into actions with minimal interference. Some of the more important design issues to be considered are: defining the building blocks for our programs and their graphical representation, specifying how they will be made available to the user, determining how to best use the limited screen space, and developing the methodology for constructing programs with our building blocks.

A. TEXT VERSUS GRAPHICS

We have used the term "visual" to mean graphics-oriented. Glinert is more specific: he defines the term visual to refer to systems whose emphasis is primarily textual but having graphical elements [Ref. 3]. Those systems that stress graphics vice textual elements are known as iconic environments. The nature of our interface will be iconic in that we intend to maximize the use of graphics to create programs, yet text will play an important role in that task. A short comparison of graphics and text is in order here. Because it is a part of the natural human communication process, pure text has certain advantages over pure graphics. However, the simple act of continuous reading and

interpretation requires concentration on the part of the user, diverting attention from the programming thought process. At the other extreme, there is a learning curve which must be overcome when using any purely graphical system. This is due in part to the ambigious nature of images, which, without amplifying information, are subject to various interpretations. A balance should be sought between text and graphics such that the best qualities of each are emphasized, and thus information transfer is maximized.

B. Target System Resources

We have stated that our interface is to be "user-friendly," and that graphics will be emphasized heavily in the programming paradigm. The target system must provide a specified minimum set of capabilities in order to support our implementation.

Graphics - There must be software routines available for drawing lines, shapes and pictures.

Mouse/Cursor - The mouse is a screen-interactive pointing device with at least one button. The operations of clicking and dragging are associated with the mouse. The cursor is the on-screen representation of the mouse movements. The cursor's appearance must be modifiable to reflect its current functionality.

Clicking - Normally this refers to the selection process where the mouse's button is pressed while the cursor is located in an object or a window. Selection is indicated by highlighting the object or activating the window.

Dragging - This is the act of selecting an object with the mouse and moving it from one location to another while holding the button down. Windows may also be dragged around the screen.

Menus - These are normally catagorized by their presentation method, which includes pull-down, drop-down, pop-up or pop-out. Menus are used to display commands, in which case the menu items are action verbs, or they may also be used for changing or toggling system attributes. Figure 2.1 shows a typical pull-down menu from a Macintosh application.

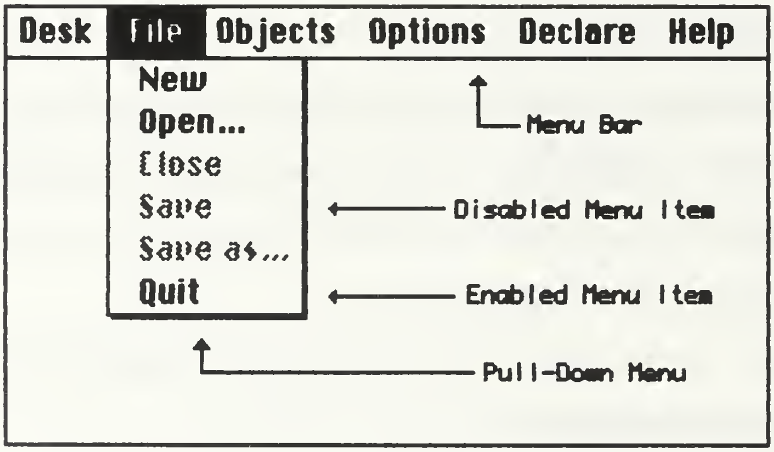


Figure 2.1 The Pull-down Menu

Keyboard - A keyboard is required for entering textual elements.

Windows - Windows are on-screen shapes, usually rectangles, in which information is displayed (Fig. 2.2). Windows must have a Size Box for resizing and a Close Box for disposing the window from the screen. The system must support multiple windows. Windows must also be movable.

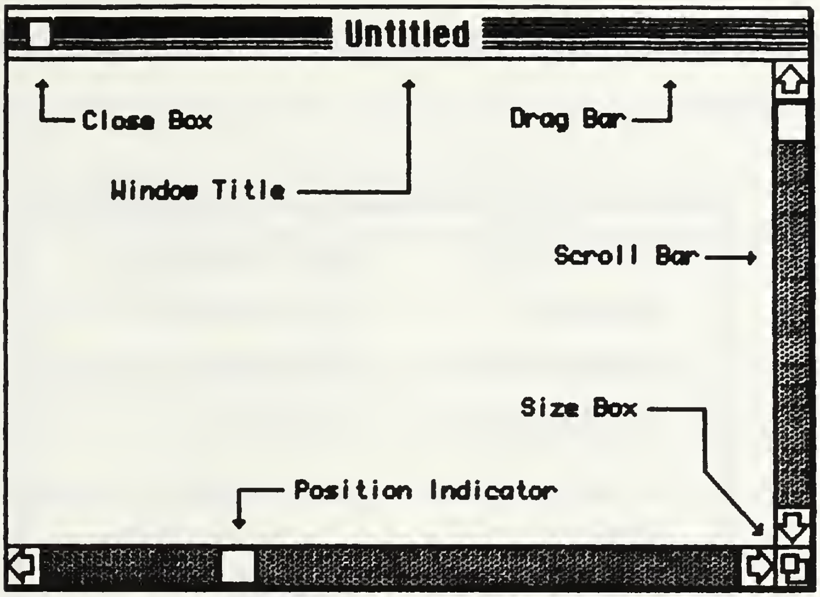


Figure 2.2 A Window

Controls - These are graphic objects that behave like physical entities when manipulated by the mouse. The system must support scrollbars for windows, pushbuttons for action initiation, radiobuttons for single selection from a set of alternative choices, and checkboxes for multiple selection from a set of choices. Figure 2.3 shows the three forms of button controls from the standard Macintosh interface.

Event Manager - In order for the user to feel in control of the system, he must be able to immediately execute commands or actions without having to perform intermediate steps. An event manager stores user-generated events from the keyboard and the mouse, as well as system-generated events from the application program. These events are then sampled by the application program using a priority scheme in order to determine the next command or action to be accomplished. The sampling process is accomplished through the use of a iterative cycle

called the event loop. Thus the user, and not the system, controls the program's actions.

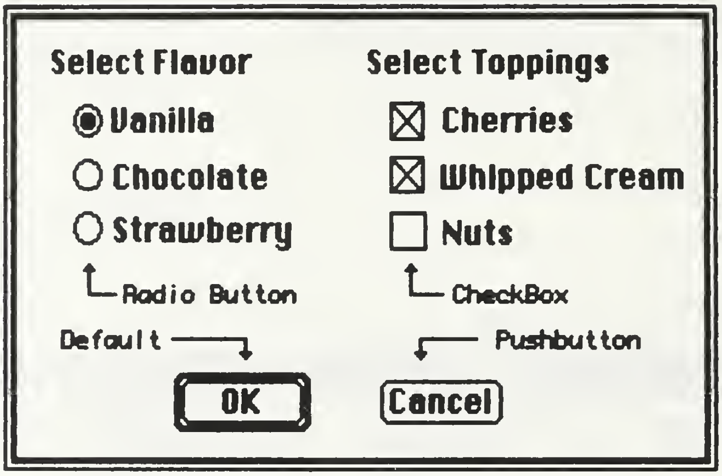


Figure 2.3 Controls

C. THE BUILDING BLOCKS

The strategy we employ to design a visual structure editor is to develop visual metaphors for both the structural and executable elements of an imperative, procedural language. These languages are composed of sets of lexical elements which are linked together according to specified syntactic rules to form statements. These statements may in turn be linked together to form programs or programmer-defined action abstractions such as the Pascal procedure or the C function. This means that our building blocks can be as small as the individual lexical elements, or as complex as the procedure or function. Since our goal is to improve performance, we want our design to strike a good balance between size and complexity. Selecting individual identifiers and operators as building blocks would make the interface cumbersome, and program construction

tedious. On the other hand, using procedure-sized units would require too much use of purely textual input, which we are trying to avoid. By choosing to use statement-sized building blocks, we can make better use of screen area as well as transferring some of the programming task from the user to the system.

In addition to the assignment statement, representatives from the various classes of language constructs are required to support the Structured Programming technology. These include: the modular construct or procedure, the definite iterative loop, the indefinite iterative loop, the if/then branch and the multiple case branch.

D. GRAPHICAL REPRESENTATIONS

Icons provide an effective means of improving interaction between the user and the computer. Icons are visual symbols representing objects or concepts. First introduced as part of the Xerox Star Information System, the icon has been widely applied in other systems such as the Apple Macintosh and Microsoft Windows. Icons may be used to represent such notions as data, data structures, operations, control and programs, but we shall use them to represent templates for the selected language constructs.

Careful planning is required to devise an iconic image which is meaningful to the user without falling prey to the tendency to pack too much information into the image. For an iconic image to be meaningful, it must be of sufficient size to be readily discernable, that is, to display distinguishing attributes that would rapidly become familiar to the user.

A reasonable set of image attributes could include the following:

* **Type** - This could be represented by a graphical image which indicates the type of language construct.
* **Location** - This could be the current position on the screen.
* **Recognizer** - This could be a modifiable text name which is initially assigned by the system.
* **Uniqueness** - This would be implicitly accomplished by consideration of the image's type, location and recognizer.

The image would then consist of an icon to represent the language construct type and a text area into which a name could be placed. The entire image must be movable (within syntactic constraints) throughout the entire program structure.

E. SCREEN UTILIZATION AND PROGRAM CONSTRUCTION

Numerous methods of representing programs have been proposed and implemented since the advent of visual programming. These include the classical flowchart [Ref. 4], structure diagrams [Ref. 5], transition networks [Ref. 6], and various hybrids of all three [Refs. 7,8]. We require a structured methodology which can be applied to a set of iconic images with which the user can build program segments rapidly without being overly concerned with program details such as declarations or parameters. The user is then manipulating the abstract form of the language construct represented by the template. The details of a particular construct are hidden from the user until such time as it is convenient for the user to return and fill in the details. In fact, some details may be filled in by the

system. While attacking this problem, we must also consider the complementary problem of screen utilization.

Intuitively, a palette seems the most efficient method of displaying the iconic images of the types of language constructs available to the user for program construction. A palette is a window in which a collection of symbols resides. Clicking on one of these symbols generally signifies intent to commence an operation or a mode change. Our palette would have not only symbols for construction, but might also contain a special symbol for accomplishing cursor mode changes (Fig. 2.4).

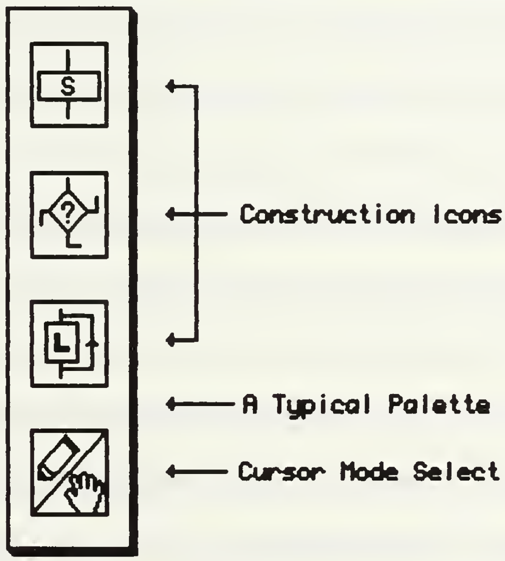


Figure 2.4 The Palette

At this point, program size is limited by the size and number of the instances of these images that could be contained in the remaining screen area. Even using the smallest meaningful representations, the screen would quickly be filled with images which would equate to a relatively

short program. The nested format of a block-structured language provides a simple, yet flexible, solution that is consistent with our desire to be able to provide an editing facility which is unhampered by screen area constraints. We use windows as on-screen repositories into which objects may be placed. These windows represent the operative statements of a language construct, that is, they are the metaphor for those statements. The on-screen analogy for nesting of statements is the ability to "open" a language construct into a window, and then that window could contain other constructs. By using either a global search facility or by following the program flow through the images and their windows, the user may easily move to a particular location in the program, much the way searching or scrolling allows localization of efforts in a textual environment.

The sequential relationships of icons in a block is expressed by connecting lines between them. A mode shift facility is required for the cursor in order to differentiate between connect operations and other operations on the icons. By clicking on the special cursor mode icon in the palette (while in the select mode), the operation of the cursor toggles and becomes connect. Visual feedback will be provided to the user so that he may always be cognizant of the functionality of the cursor.

III. IMPLEMENTATION

Our goal is to pursue implementation of the prototype to the point of allowing the user to manipulate iconic images representing language constructs, thus building a program from these images, and then to provide feedback in the form of textual output to the screen or to a printable file on secondary storage. This paradigm is based on constructs which are a subset of the Pascal language being stored as Objects in a database. We use the common term Object to refer to the iconic images, and the abstract database which represents their relationships is called the Object Tree.

Pascal was selected as the language of choice because of its readibility, portability and conformability to the discipline of Structured Programming. The syntax of the selected language subset is depicted in Appendix A.

The Apple Macintosh was selected as the target machine mainly due to its rich selection of built-in software with which a programmer can easily create the facilities of a user-friendly environment.

A. DEFINITIONS

In order to adequately describe the operation of the interface, we must specify some terms.

Connecting - Drawing a connecting line between the two Objects, or between an Object and the top or bottom of the screen.

Buildwindow - A window in which Objects may be dragged, dropped and connected for the purpose of constructing program segments.

Active Window - This is the Buildwindow which is currently active. Activation is made apparent to the user by the window's scroll bars becoming visible.

Inactive Window - Any visible window that is not the Active Window.

Text Window - A text window displays the textual results of Object manipulations.

Dialog Box - A special kind of window used for soliciting or conveying important information, dialog boxes are used for error messages, data input and file operations.

Parameter Box - This is a dialog box which is used for entering textual elements associated with a particular Object.

B. THE OBJECTS

This prototype is an experiment in using graphical manipulations to construct programs and is therefore not designed to accomodate all the facilities of the entire language. Representative language constructs from the various classes are provided to support the basic elements of Structured Programming. The Object images are depicted in Appendix D. We now specify their equivalent language constructs in terms of the Pascal subset.

Program - The only Object not represented by an iconic image, the Program represents the Pascal program, and is manifested by a window which is opened during system initialization.

Sequencer - The generic term for Objects that are equivalent to simple sequential steps in a program. The Object types derived from the Sequencer are the Procedure, the Block and the Basic.

Procedure - The basic modular construct, Procedures are declarations, may only be dropped within the Program and other Procedures, and are not connectable. The Procedure enables nesting of other Procedures and Calls, as well as other Objects.

Call - The instance of a Procedure, a Call is created from a Procedure and then moved to the location from which the Procedure will be called. Calls are elemental units of the Object database.

Block - A special construct used to extend the maximum allowable number of Objects contained in a Buildwindow, the Block is the only Object which does not represent a template for a language construct.

Basic - This is the elemental sequential unit from which programs are built. Each Basic contains a series of Assignment or Input/Output statements. Basics are also elemental units of the Object database.

Branch - The generic type associated with branching control constructs in a program, the two Branch types are the If and the Case.

If - The basic branching construct, the If is associated with either one or two Buildwindows, depending on whether or not the Else part is required.

Case - The multiple branch construct, the Case has a Buildwindow associated with each declared Case Constant.

Loop - This is the generic type which represents the iterative type constructs, the For and the While. Only one Buildwindow is associated with this Object class.

For - The definite iterative construct.

While - The representative conditional iterative construct.

C. THE OBJECT TREE

The Program is initialized as the root of the Object Tree. As new Objects are dragged onto the Active Window, they are added to the Object Tree. The abstract structure of the Object Tree is basically represented by two associated tables. After a successful drag has been made, a new entry is made in the Objects table, and the containment relationship is recorded in the Has Objects (HO) table. Each Object type has a direct relationship (HO table entries) between itself and its contained objects except for the Branch class of objects. Because of the requirement to divert control flow, a design decision was made to create separate Objects for each possible path of execution. Thus, for the If, two additional entries are made into the Object table with special type identifiers to specify them as Then and Else siblings, respectively. All further containment relationships are then recorded with the appropriate sibling (Then or Else) in the HO table. The Case situation is similar, with four new Objects being added to the Object table, one for each potential case constant. This abstract structure allows Objects to be nested to any

level, and allows for a relatively simple tree traversal to retrieve the data. The leaves of the tree are the Basic and the Call Objects.

D. THE SHELL

In order to preserve the programs created with the system, it is built around a shell from which programs may be saved or retrieved from secondary storage. With the system in operation and in the Null state, the user is be able to select and open a program from a list of stored programs, or open a new Program template, in either case transitioning to the Active state. From the Active state, the user has the following options: to save a program, to save a program under a different name while retaining the original version, to close a program and transition to the Null state, or to quit the application entirely. The system itself may be started in one of two ways: a program document may be selected and opened, loading the subject program onto the shell and placing the system in the Active state; a new program template may be initialized and loaded. During transition to the Active state, the Program window is opened and the system is also placed in the Select mode (Fig. 3.1).

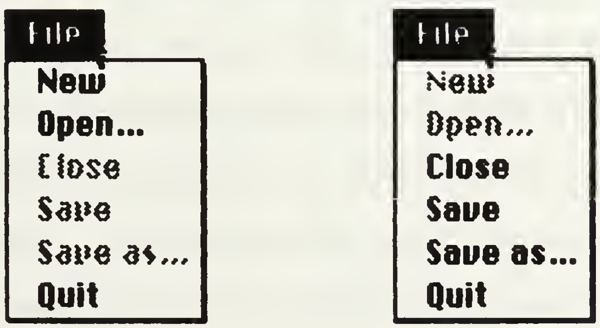


Figure 3.1 The File Menu for Null & Active Shell States

E. CONSTRUCTING PROGRAMS

Programs are constructed through a combination of Palette operations, menu operations and screen operations. These operations may result in requests for information from the user, which are handled through the dialogs and parameter boxes depicted in Appendixes F and G.

1. Palette Operations

The Palette is a permanent window located on the left side of the screen. In addition to the decorative application icon at the top, the Palette contains four other icons. The middle three are used for program construction. They represent the generic icons for the three basic classes of constructs, the Sequencer, the Branch and the Loop. The bottom icon is the Mode Select, and is used to shift between the cursor modes of Select and Connect.

While in the Select mode, new Objects may be dragged from the Palette to the Active Window. By clicking in one of the three Object icons in the Palette, the cursor changes appearance to signify commencement of the drag operation. The Mouse is then dragged until it passes over the boundary of the Active Window, at which time a dotted outline becomes visible, signifying that the Mouse is in the allowable drop zone. The initial drop of an Object is restricted to the Active Window to prevent inadvertant drops in the wrong window while several windows are open on the screen simultaneously. An inactive window may be activated by simply clicking in it. When the mouse button is released, the drop location is verified, and if syntactic criteria are met, the Object is drawn in the window, and its record is added to the Object Tree database.

Figure 3.2 The Palette and the Program Window

Information regarding the choice of a specific type is then solicited from the user. The user may make a choice then, or delay his choice until such time as he is more certain of his programming requirements. Initially, a name is assigned to each Object consisting of the prefix 'Obj' concatenated with the Object's unique identification number. This is the Recognizer attribute, which may be retained as is, or changed at the discretion of the user. The Object's icon delineates its specific type.

At this time, allowable menu operations become enabled. These Menu operations are dependent on the Object's type and connection status. For

example, if a new Sequencer is dragged to the Active Window, appropriate commands are enabled in the Objects menu, including Select Type (Figure 3.3). If the Sequencer's type is changed to Procedure, then the menu will be changed to reflect the allowable operations (Figure 3.4).

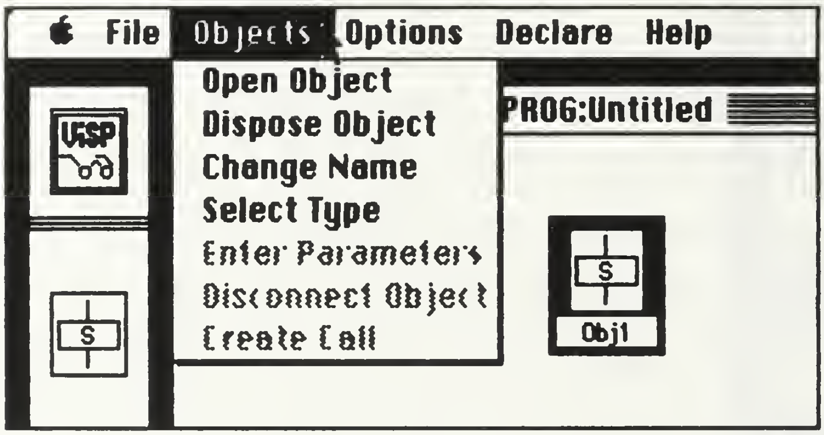


Figure 3.3 Objects Menu Operations for a Sequencer

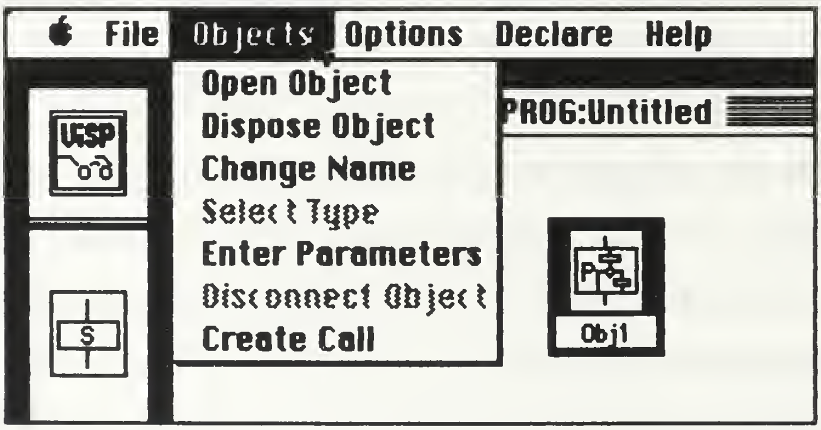


Figure 3.4 Objects Menu Operations for a Procedure

To shift to the Connect mode, clicking in the Mode Select icon while in the Select mode will change the appearance of the cursor to a pen, thus

indicating the user's ability to connect objects to each other or to the top (the "begin") or bottom (the "end") of the screen. The system will only allow a maximum of two connections per Object, and will only allow one "begin" and one "end" to be established. If a connected Object is selected, Disconnect Object becomes enabled in the Objects menu (Figure 3.5).

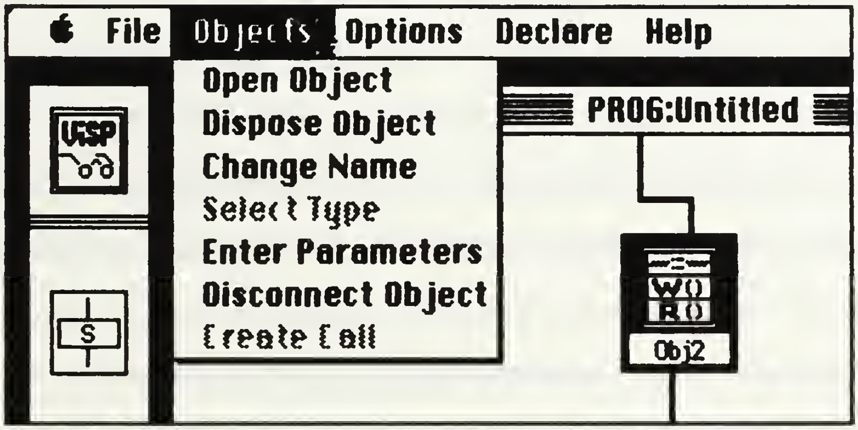


Figure 3.5 Disconnect Object from Objects Menu

2. Screen Operations

Already established Objects which are not connected may be moved within a Buildwindow to an empty location, or to replace another previously established Object, or may be moved to empty locations in other open Buildwindows (if such a move would not violate the program structure). Moves are initiated by clicking in and dragging the Object to the desired location. Replacements in other than the Active Window are not allowed in this situation. Since an established Object is being dragged, the cursor's appearance will differ from what it would be if a new Object were being dragged.

3. Syntax Enforcement

Syntactic constraints are enforced by restricting Palette operations, screen operations and menu operations. If an action is an obvious syntax violation, for example, attempting to nest a Procedure declaration within a For loop, then the action is disallowed, in that particular instance being accompanied by an error message. The restrictive mechanism is consistent. If a menu operation is not allowable, then the equivalent command from the menu will be disabled. Unauthorized screen operations result in one of two alternatives. If the inconsistency happens at the outset, say, trying to start a connection in a Procedure, then the action is ignored. If the error is at the end of the operation, say, trying to end a connection in an Object which already has two connections, then the action is disallowed with an error message to indicate why the operation was not completed. The System error messages are listed in Appendix E. Finally, Palette operations which may result in syntactic errors are simply disallowed.

F. MENUS

The menu bar is located at the top of the screen. The menus are titled according to the types of commands which they represent. Dialog boxes are associated with various menu commands. Parameter Boxes from Objects and Declare commands are used to enter information.

1. The File Menu

This menu contains the commands which access new and old programs. Selection of menu items from the File menu allow transitions between the Active and Null Shell states.

New - Enabled only when the application is in the Null mode (no program being edited), New initializes data structures and opens a new program for editing.

Open - Also enabled only from the Shell, Open presents the user with a list of saved programs from secondary storage that may be opened for editing. The user is allowed to select and open, or cancel the command.

Close - Enabled when a program is active, Close transitions the application to the Shell mode. If a change has been made to the current active program, prompts user to Save changes.

Save - Enabled when a change has been made to the active program, Save rewrites the formatted program to secondary storage.

SaveAs - Always enabled, SaveAs prompts the user for the new name or cancel, and also prompts to replace if that file exists.

Quit - Always enabled, Quit prompts to save changes if any, then terminates application, transitioning to the Macintosh file system.

2. The Objects Menu

The Objects menu contains commands which initiate operations on the Objects. Generally, an Object must be selected to enable these commands.

Open Object - Enabled only when an Object has been selected, opens Objects with block attributes into Buildwindows, opens Basics and Calls into the associated Parameter dialog box. The If and the Case are special cases. After selecting Open Object, a small dialog box is presented to the user soliciting the user's choice of the particular buildwindow to be opened on the screen. The If object allows for two possible choices, the Then window and the Else window. The Case object has four allowable cases which open up into windows for additional program construction. The Basic object does not open for further construction, but rather forms the basis for the recursive structure of the program. Basics open up into dialog boxes into which up to five executable statements may be entered.

Dispose Object - Enabled only when an Object has been selected, recursively disposes all the Object's descendents and then disposes the Object. If the selected Object is a Procedure, also disposes any associated Calls. If the selected Object is an If or Case, associated sub-blocks are also disposed.

Change Name - This is enabled when an Object has been selected, except for a Call, whose name remains the same as its parent Procedure. Presents the user with a text entry box (Fig. F.2) in which to enter the new name or cancel.

Select Type - This is enabled when a generic Object has been selected. Allows for selection from a list of available types within the language construct class, e.g., Branch may be changed to If or Case. Some type selections may be restricted for syntactical reasons, e.g., Procedures

may only be declared in the Program window or other Procedures, so selection is disallowed in other type windows.

Enter Parameters - This opens the appropriate dialog box into which the user may enter specific textual data concerning the Object.

Disconnect Object - Selecting Disconnect Object removes all connections from the selected Object.

Create Call - This command creates an instance of a Procedure which can be dragged to any Buildwindow from which the parent Procedure is "visible."

3. The Options Menu

This menu is for items that are not directly associated with program construction. It contains commands for text display and also those which control application program options (Fig. 3.6).

Select Type Dialogs On/Off - controls the automatic appearance of the Select Type Dialog box which appears when a new generic construct is dropped in the Active Window.

Show Text On/Off - determines whether or not a text window is displayed on the screen when the Print Program or Print Object commands are selected.

Print Program - This command writes the textual version of the current form of the program under construction.

Print Object - This command writes the textual version of the selected Object's abstract structure.

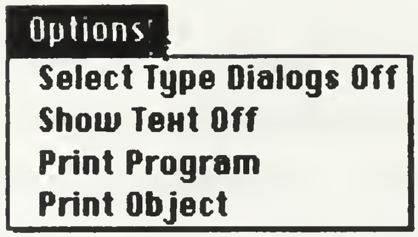


Figure 3.6 The Options Menu

4. <u>The Declare Menu</u>

This menu contains commands which bring up dialog boxes into which global and local declarations may be entered (Figure 3.7). Locals may only be entered by selecting a Procedure.

Global Constants - Enter Program constants.

Global Types - Enter Program types.

Global Variables - Enter Program variables.

Local Constants - Enter selected Procedure's constants.

Local Types - Enter selected Procedure's types.

Local Variables - Enter selected Procedure's variables.

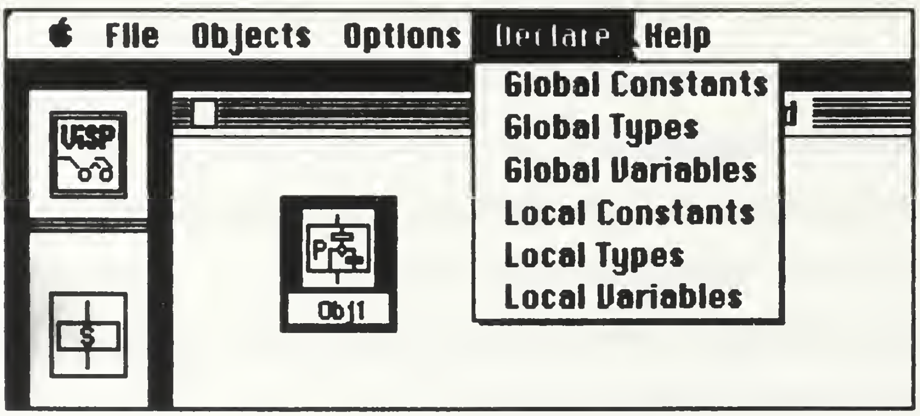


Figure 3.7 The Declare Menu

5. The Help Menu

The Help menu provides brief descriptions of the Palette icons and their representations (Figure 3.8).

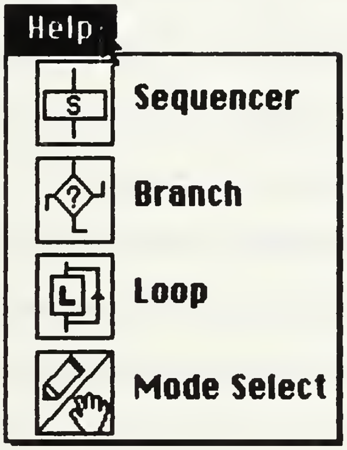


Figure 3.8 The Help Menu

IV. CONCLUSIONS

A. DISCUSSION

1. The Interface

With a minimum of practice, a user can rapidly assemble the components of small programs. The system is easy to use, and most actions are intuitive and natural. The iconic images are simple and are easily recognized. Particularly useful are the "Select vice Enter" features implemented in support of the user-friendly principles. These features allow the user to select from a list of available choices rather than having to remember those choices.

Although the graphics produced by the Macintosh's internal software are superb, screen area is still a problem, due in part to the small size of the Mac's screen, and compounded by the large size of an Object's image. Less clutter can be realized by reducing the size of the image and incorporating one of the new page-sized screens which are now available for the Macintosh.

The Macintosh User Interface guidelines also specify a double-click (two clicks within a short, specified time period) action as equivalent to an open action command. That shortcut would have been helpful in reducing the number of times that we used the menu for entering parameters.

The most severe problem of the system, which is a fundamental problem in all visual programming systems, is its inability to adequately

visualize locality. While working within several nested layers, it is possible for the user to mentally lose track of the surrounding context. A solution to this problem is a global locating facility, which could display a scrollable listing of the Object's names. This display could be somewhat miniaturized, with the Objects ordered in sequence as they are in the program.

2. Extensions

Several facilities could be added to the system which would facilitate more rapid programming. These are a Duplicate Object command to create exact replicas of Objects, and a Store Object command to store copies of often used Objects in a user-defined library. These stored Objects could then be retrieved through a selection process similar to opening a program. Another necessity is a Search function. Searches could be conducted using either a name or type. The designated start point for the search could be indicated either by selection or defaulting to the entire program.

The original concept for the interface envisioned a small parser for expressions and statements entered into the parameter boxes. Identifiers parsed from these entities would then be entered into a table which would be accessible by the user to remind him which entries required resolution. Constants, types or variables could then be automatically entered into the appropriate declaration boxes.

The language constructs chosen to be represented as Objects were chosen based on their syntactic requirements and structure. Expansion to include the entire Pascal language can be accomplished by using the design

principles employed herein. As an example, a Function could be declared exactly as is a Procedure, and then calls to that Function would be acceptable in any Object where they were lexically and syntactically correct.

The graphical appearances of the Object images do not reflect any great insight into iconic representation, but rather are just simple representations that seemed fairly familiar and easily recognizable. By using the Macintosh as our target system, we are able to store these iconic representations such that we can modify their appearance without having to change the application program. Similarly, the appearance and labelling used in the dialogs and error messages may also be rearranged or changed. There are several public-domain tools which may be used to make these modifications, and it seems that incorporating such a tool into the interface would give the user the opportunity to fashion the interface to conform to his own personality, thus taking another step toward increasing programmer productivity.

The Object database is composed of several tables. A program's abstract form is not a conventional tree but is realized through the table's relationships. To facilitate adding an interpreter or incremental compiler to this interface using current technology, the system would most likely have to be modified to store the program in the classic tree form.

B. SUMMARY

The objective of this research was to develop a visual programming paradigm which incoporated a user-friendly interface. A peripheral goal

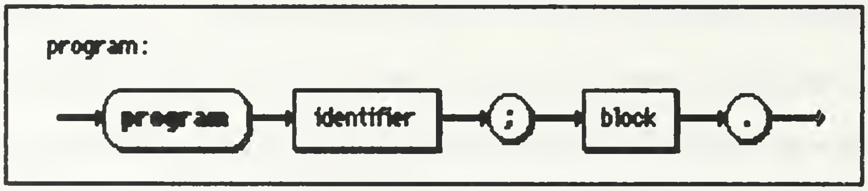
was to determine what features would be necessary to achieve the most effective interface possible. Our primary objective was achieved. We designed and implemented a prototype visual programming system integrated in a user-friendly interface. The prototype has been used to create small programs which have subsequently been compiled successfully. Through the addition of the facilities discussed above, the prototype could be extended to handle much bigger programs with greater efficiency.

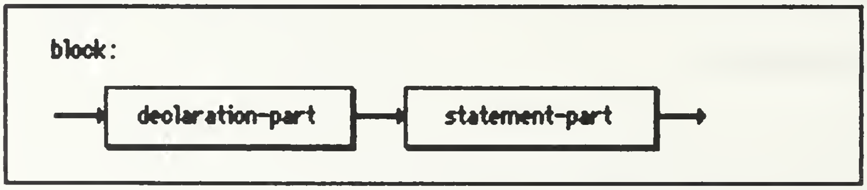
As with any system which is built from scratch, the transition from to implementation was not without tribulation. The learning curve associated with mastering the Macintosh graphics routines and managers is steep, requiring many hours of practice to determine the visual effects produced through various resource combinations. Once that obstacle is overcome, the vast amount of commercial and public-domain software tools available for the Macintosh makes it an excellent system on which to do development.

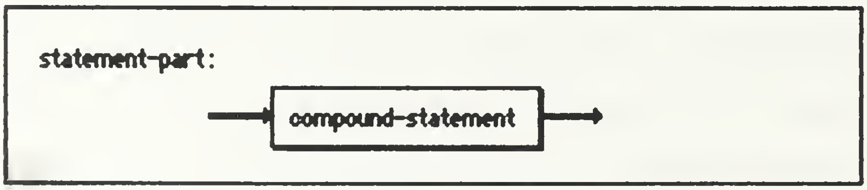
APPENDIX A

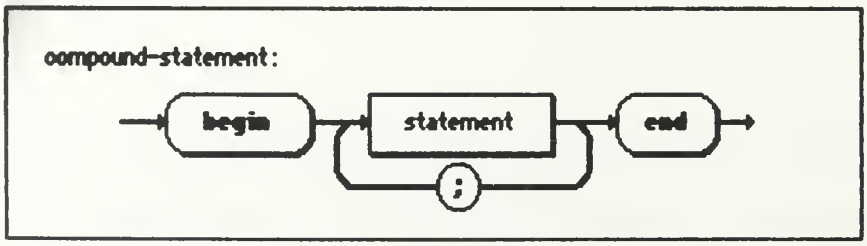
SYNTAX

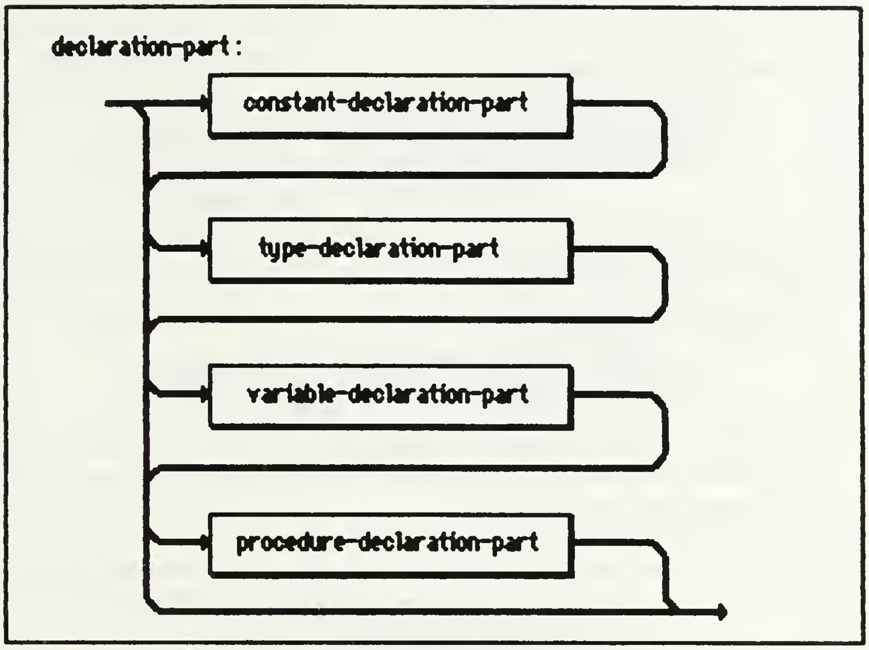
The syntax described here is not meant to represent the entire Pascal language, but merely the subset associated with the Objects used for program construction. Major constructs not supported by the system are: labels, goto statements, with statements, repeat statements and functions. The definitions for identifier, expression, constant, type and variable should be those of the follow-on compiler or interpreter. Terminals are indicated by bold text displayed in ovals.

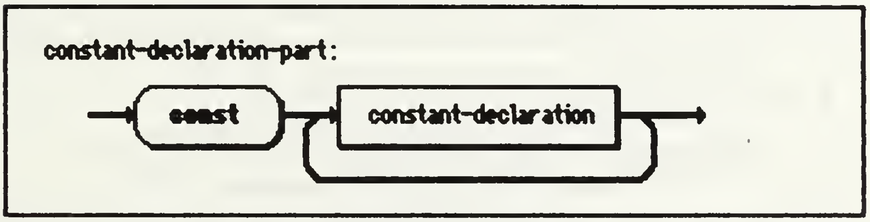


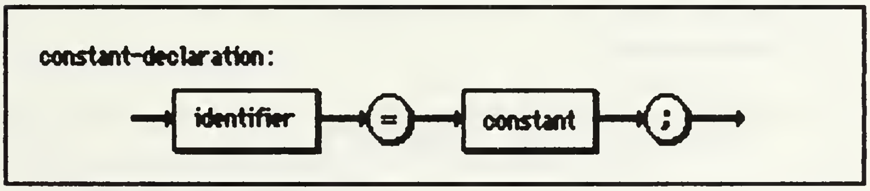


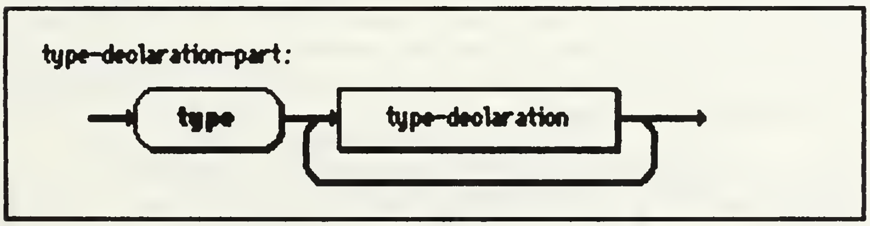


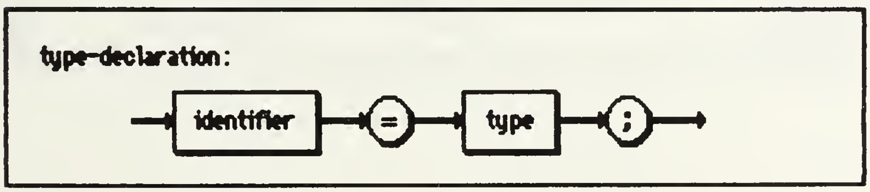


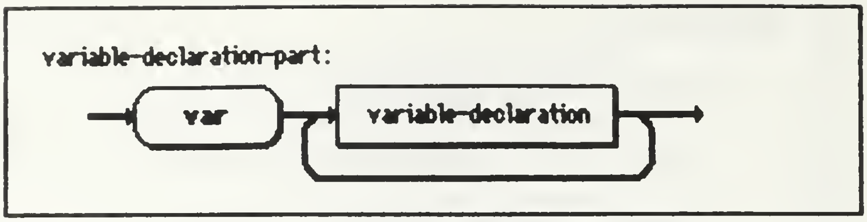

declaration-part:
constant-declaration-part
type-declaration-part
variable-declaration-part
procedure-declaration-part

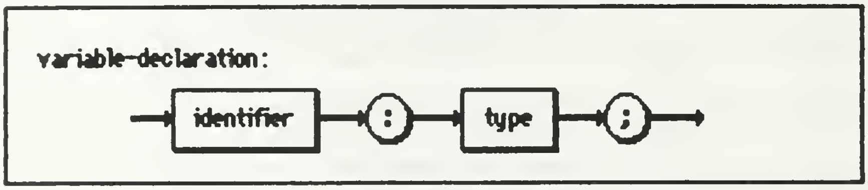

constant-declaration-part:
const
constant-declaration

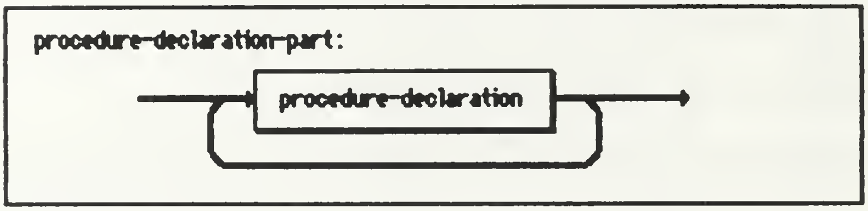

constant-declaration:
identifier
=
constant
;

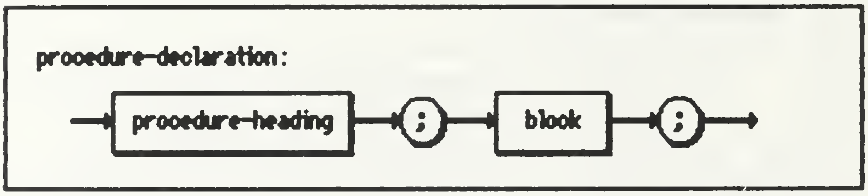

type-declaration-part:
type
type-declaration

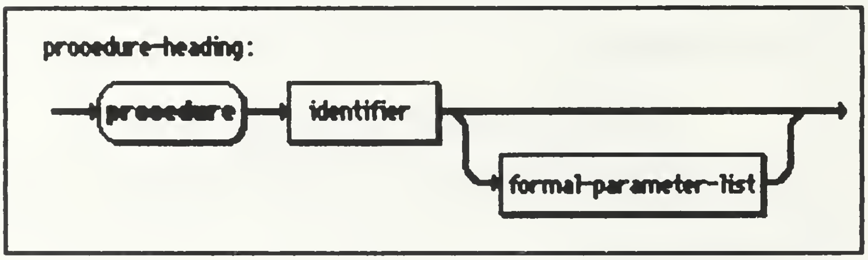

type-declaration:
identifier
=
type
;

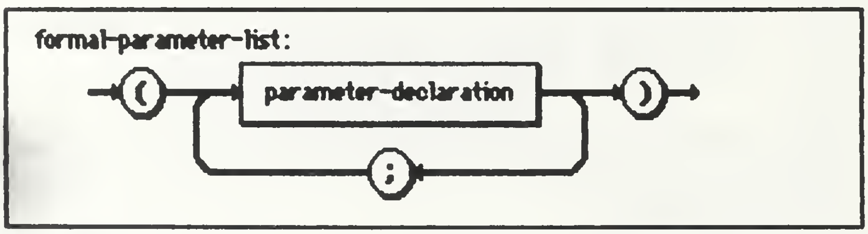

variable-declaration-part:
var
variable-declaration


variable-declaration:
identifier
:
type
;

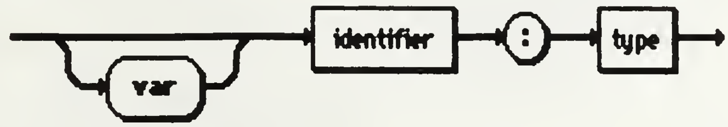

procedure-declaration-part:
procedure-declaration


procedure-declaration:
procedure-heading
;
block
;

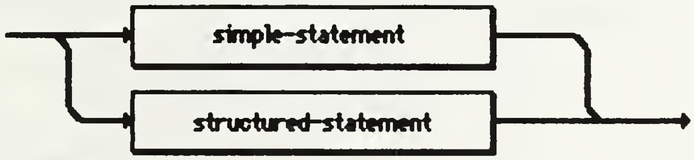

procedure-heading:
procedure
identifier
formal-parameter-list


formal-parameter-list:
(
parameter-declaration
)
;

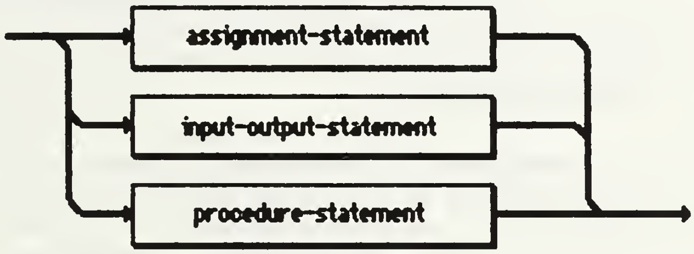
parameter-declaration:



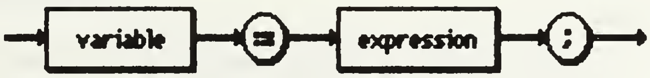
statement:



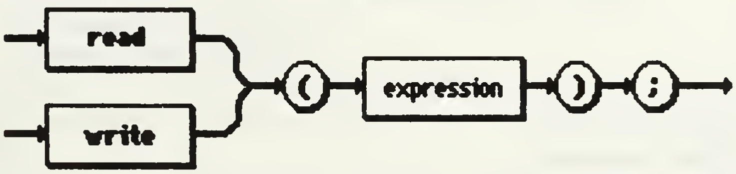
simple-statement:



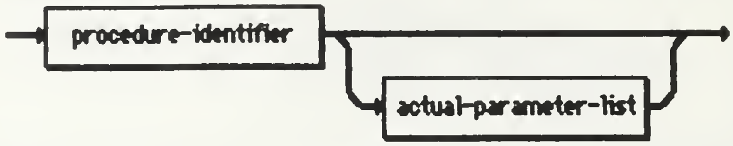
assignment-statement:



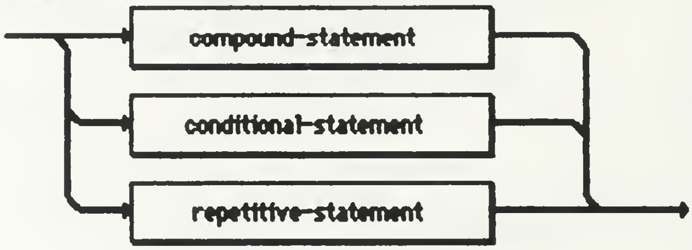
input-output-statement:



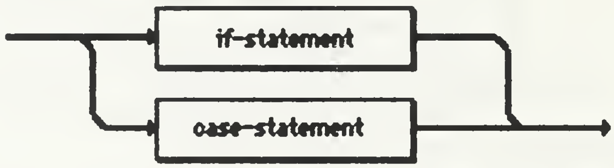
procedure-statement:



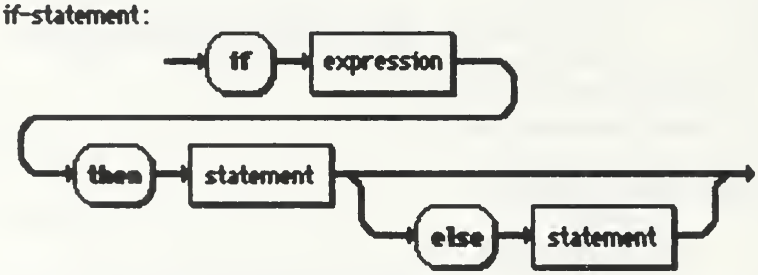
structured-statement:



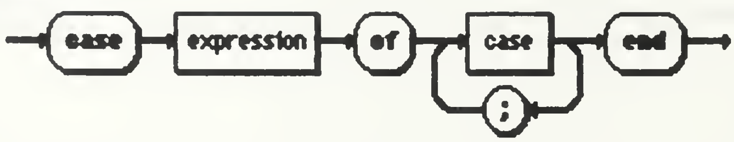
conditional-statement:

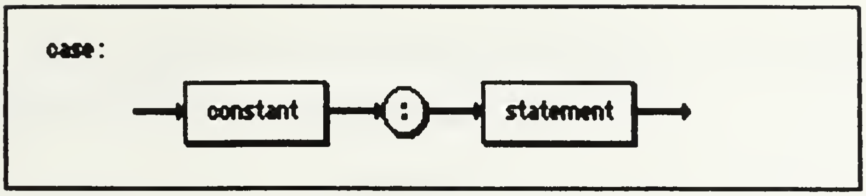


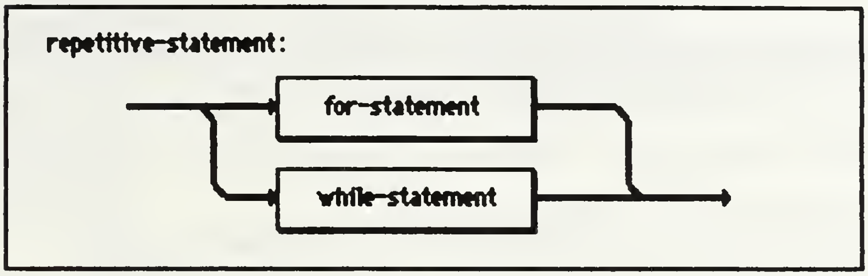
if-statement:

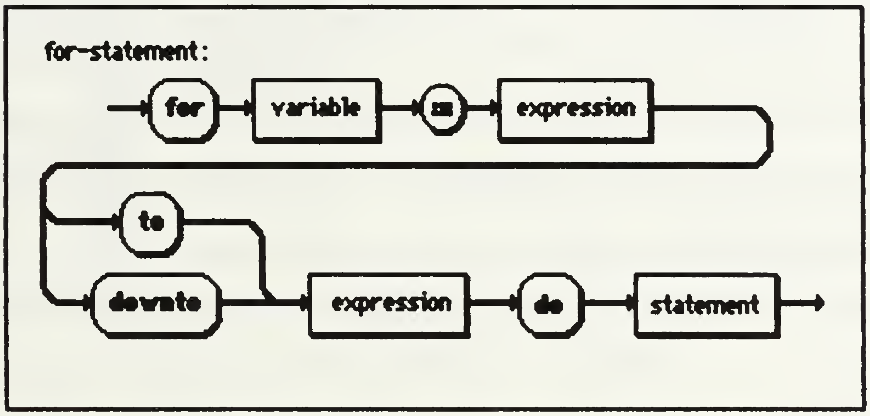


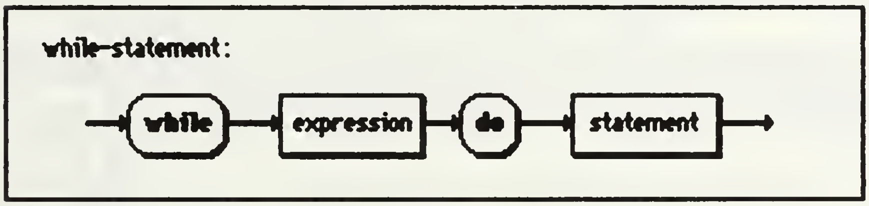
case-statement:




case:
constant
:
statement


repetitive-statement:
for-statement
while-statement


for-statement:
for
variable
:=
expression
to
downto
expression
do
statement


while-statement:
while
expression
do
statement

APPENDIX B

TUTORIAL

This brief tutorial is intended to give the user a feel for the mechanics of program construction. Parameter Boxes and other dialog boxes not shown are depicted in other Appendixes.

After the application is started, the Program window is opened for construction. We start by naming the program. Pulling down the File menu, we release the mouse button when Save As is highlighted, bringing up the Save As Dialog box, into which we will enter the name of our example program (Fig. B.1).

We wish our program to take a number from the keyboard, do some computations on the number, and then print the number to the screen. By separating the problem into three separate tasks, we exercise the principle of division of labor, thus making the problem easier to understand and program. We shall name the three tasks as follows: Initialize, Compute and Output. The procedure is the modular construct in Pascal which allows this kind of sub-tasking. To declare the procedures which will accomplish our tasks, we start by dragging a Procedure Object to the active window (Fig B.2). The new Procedure is then renamed to "Initialize" by selecting Change Name from the Objects menu. A Call to the Procedure is then created by selecting Create Call from the Objects menu, and moved into position for connection (Fig. B.3). Similar actions are required to declare and create Calls to procedures "Compute" and

"Output". These Calls are then connected in the order in which they are to be executed (Fig. B.4).

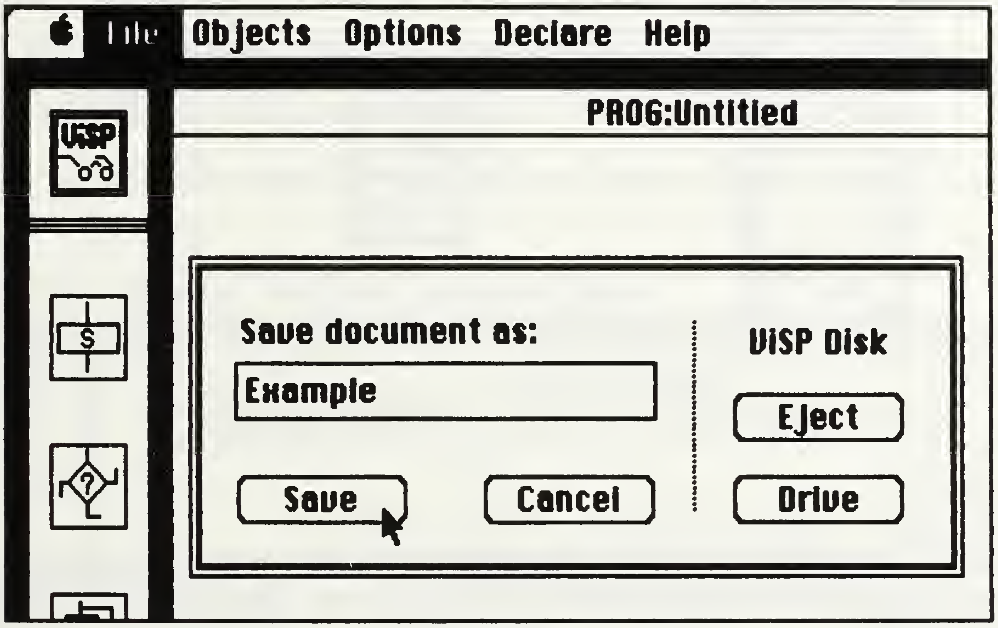


Figure B.1 Naming the Program

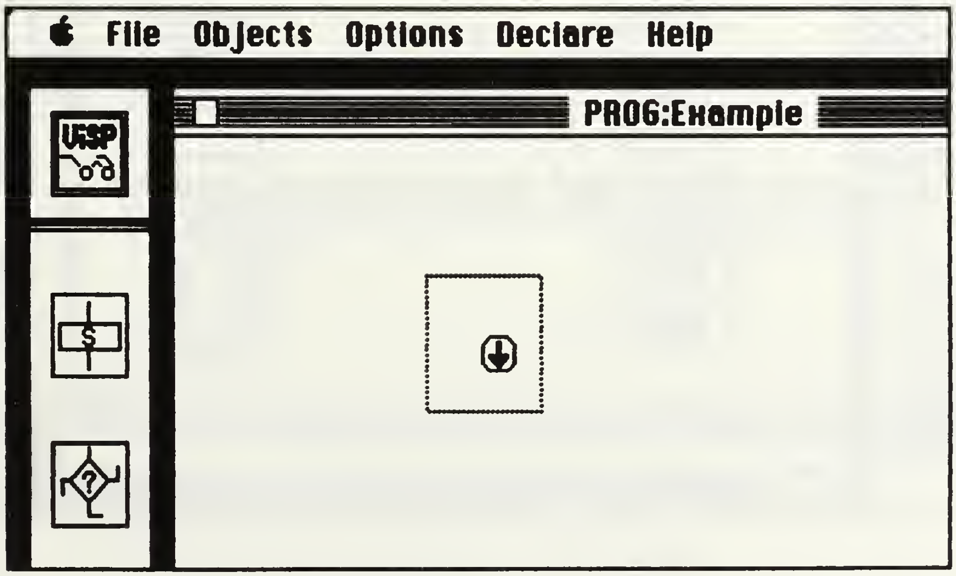


Figure B.2 Dragging a New Object

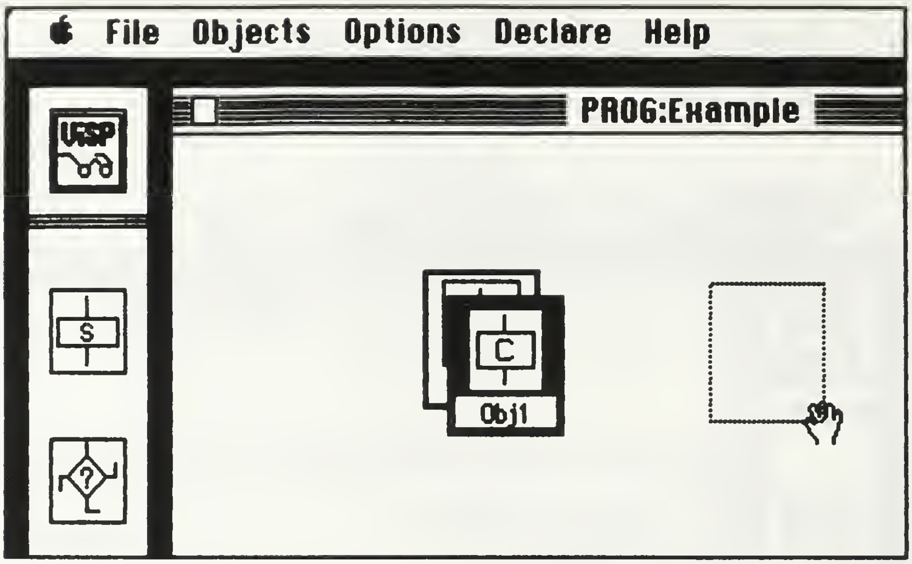


Figure B.3 Dragging the Call

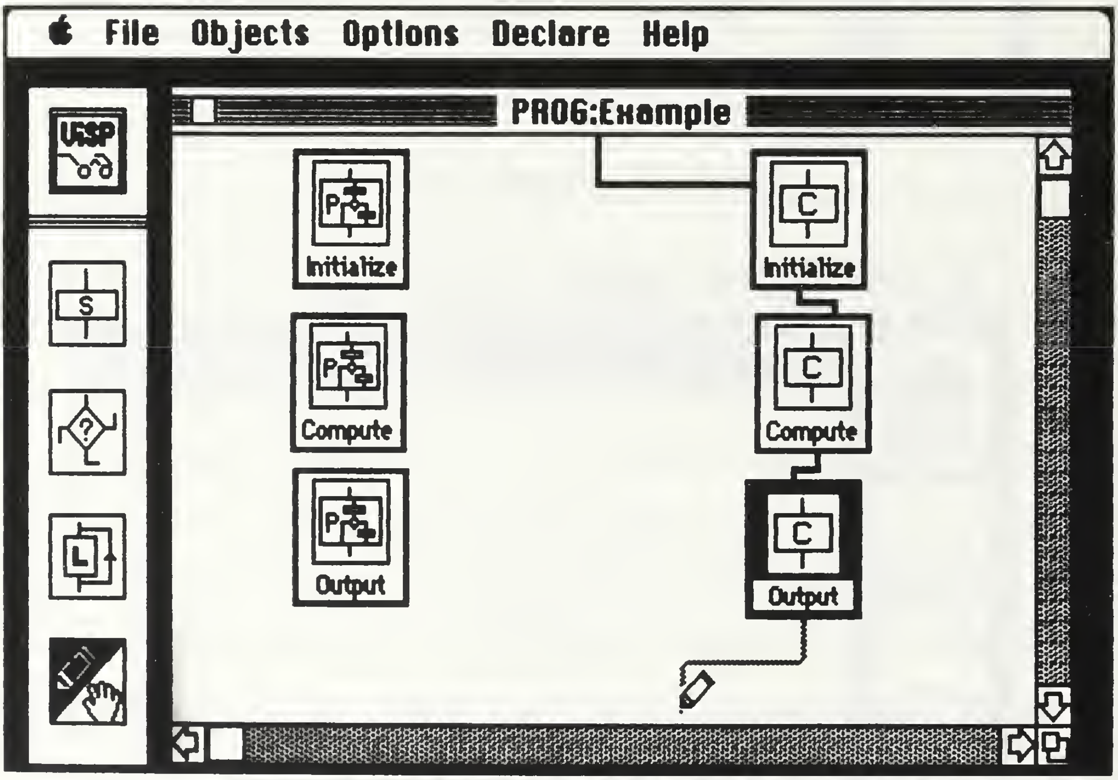


Figure B.4 Connecting the Calls

Procedures "Initialize" and "Output" are constructed by opening each Procedure's Buildwindow, and then dragging a Basic into the open windows. After the Basics are then connected properly (Fig. B.5), we select Enter Parameter from the Objects menu, and enter the appropriate I/O statement. By selecting Print Object with Procedure "Initialize" highlighted, we may view the intermediate results of our efforts (Fig B.6).

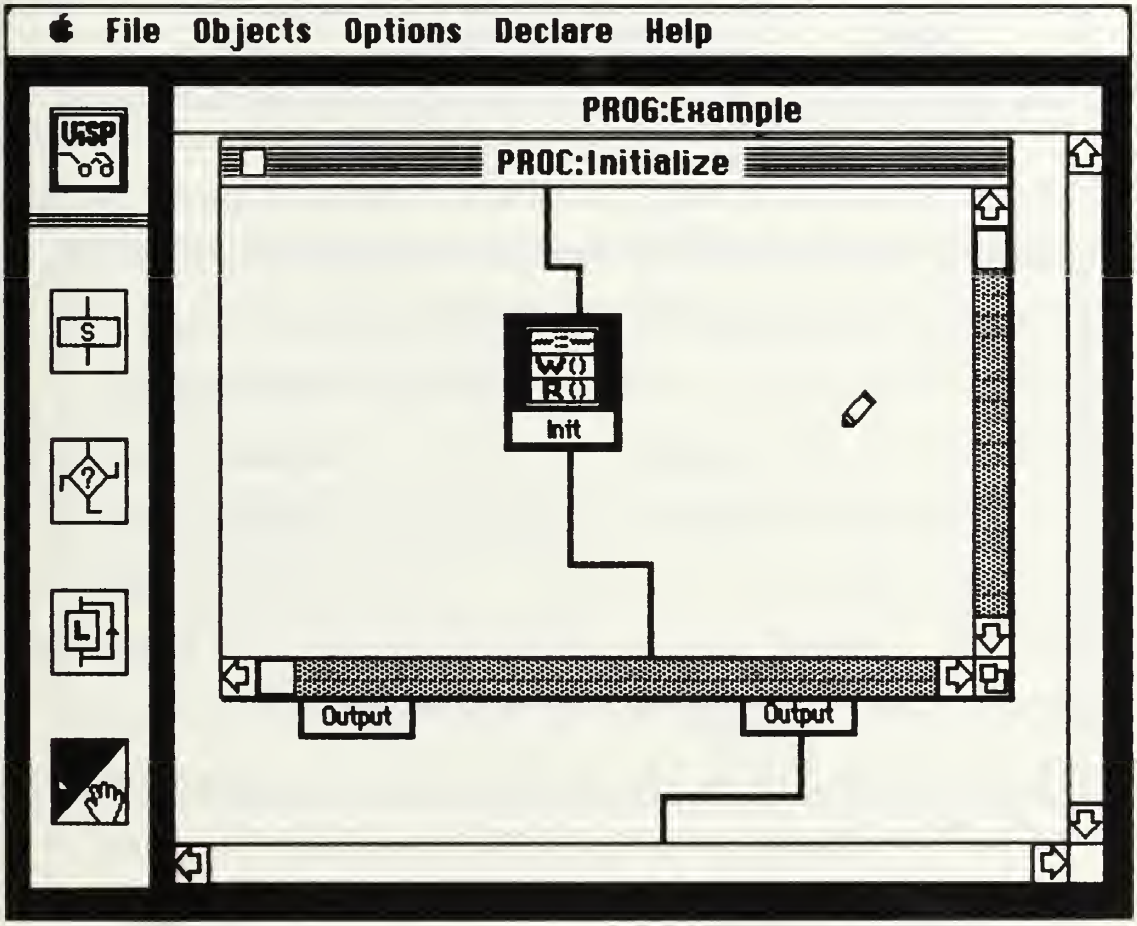


Figure B.5 Connecting the Basic

Having completed the first and last procedures, we now must construct the "Compute" procedure. There will be an additional compution necessary,

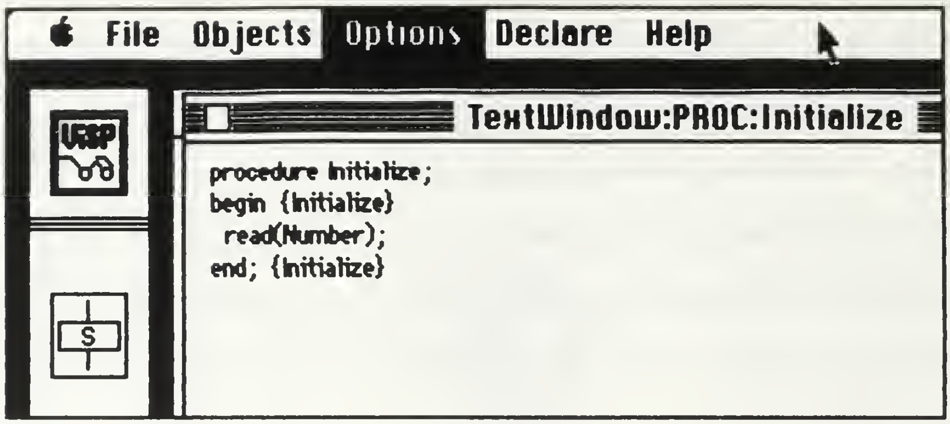


Figure B.6 Print Object "Initialize"

so we will again delegate a sub-task to demonstrate the use of a nested procedure. After the "Compute" Procedure's window is opened, the new Procedure is declared and named "Product," a name indicative of its function. Formal parameters are required and declared through a parameter box (Fig. B.7). A Call is created for "Product" and the Procedure's "var" and "type" formal parameters are echoed in the Call's actual parameter box (Fig B.8). The remainder of "Compute" is now constructed using a Basic and a While. The While is opened, and the Call to "Product" is dragged from its declaration window to the opened While window (Fig. B.9). Constants and variables are then added using the Declare menu. The program may then be viewed in its textual form by selecting Print Program from the Options menu (Fig. B.10). Since the program text is larger than the text window, pressing any key will stop the scrolling text, as well as resume scrolling. The entire program is also written to a text file (Fig. B.11) that may be accessed for printing, interpreting or compiling.

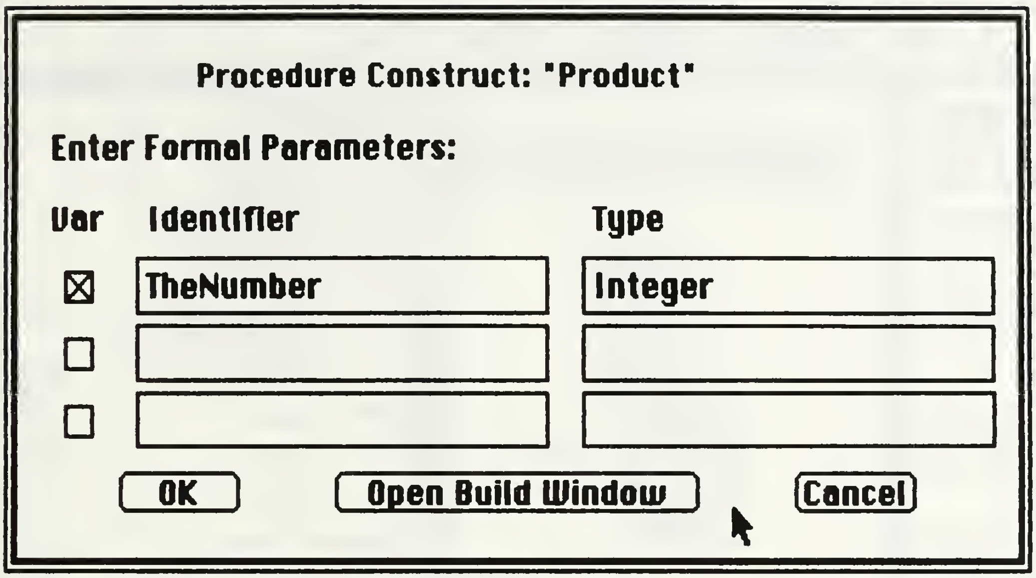


Figure B.7 Entering Formal Parameters

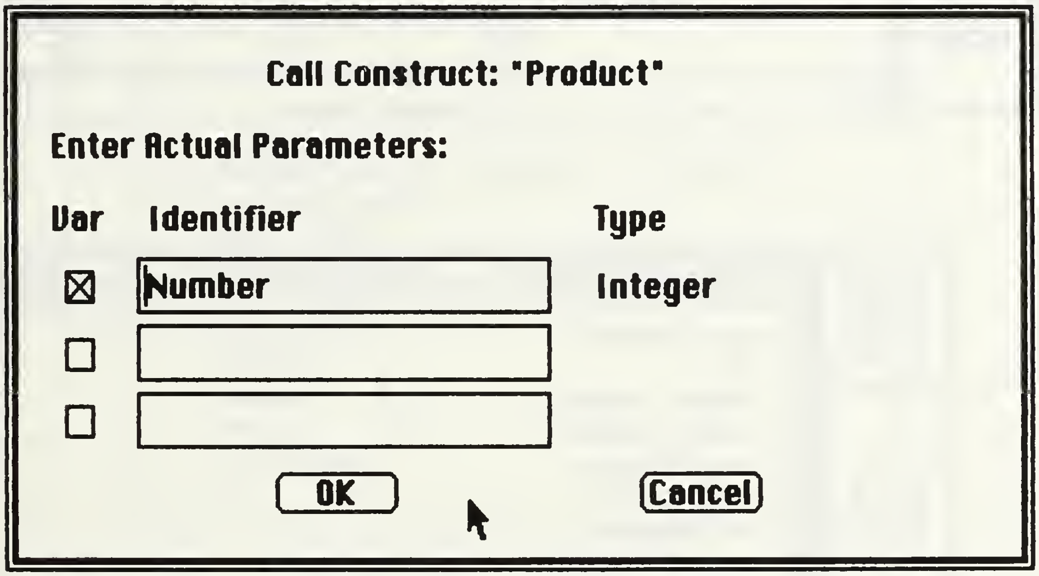


Figure B.8 Entering Actual Parameters

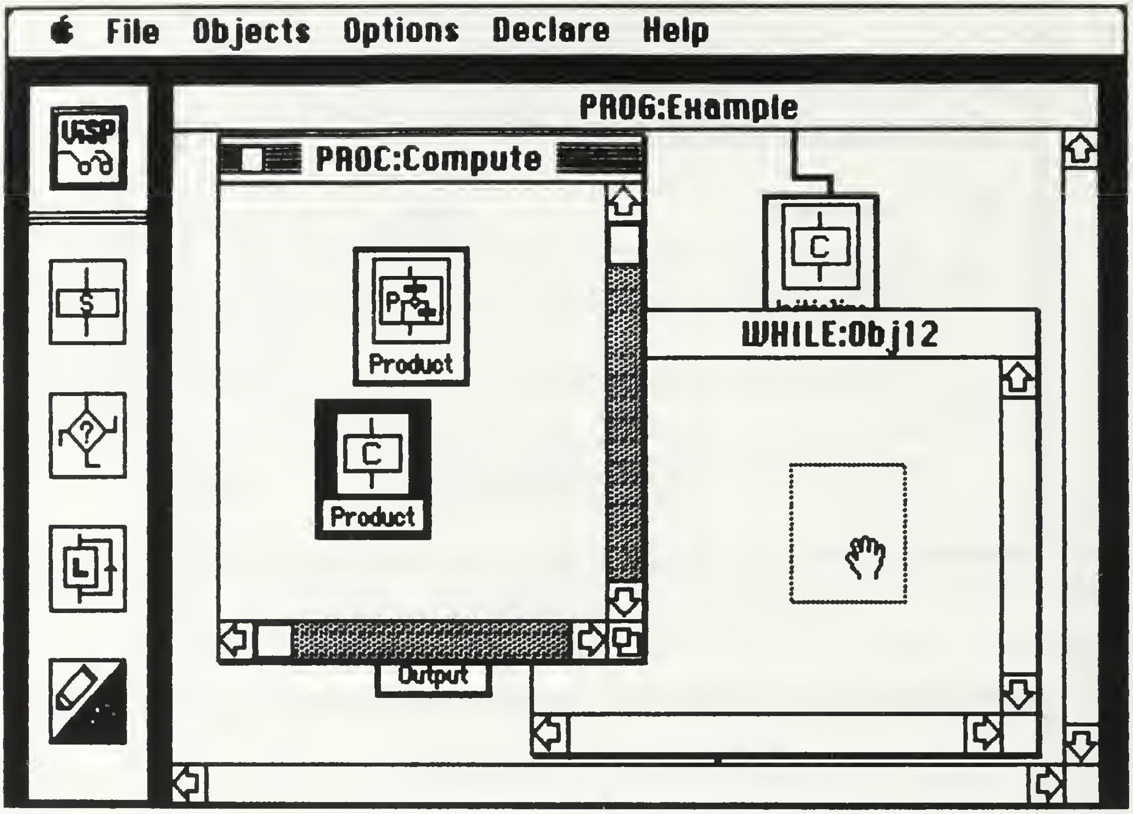


Figure B.9 Dragging to Other Windows

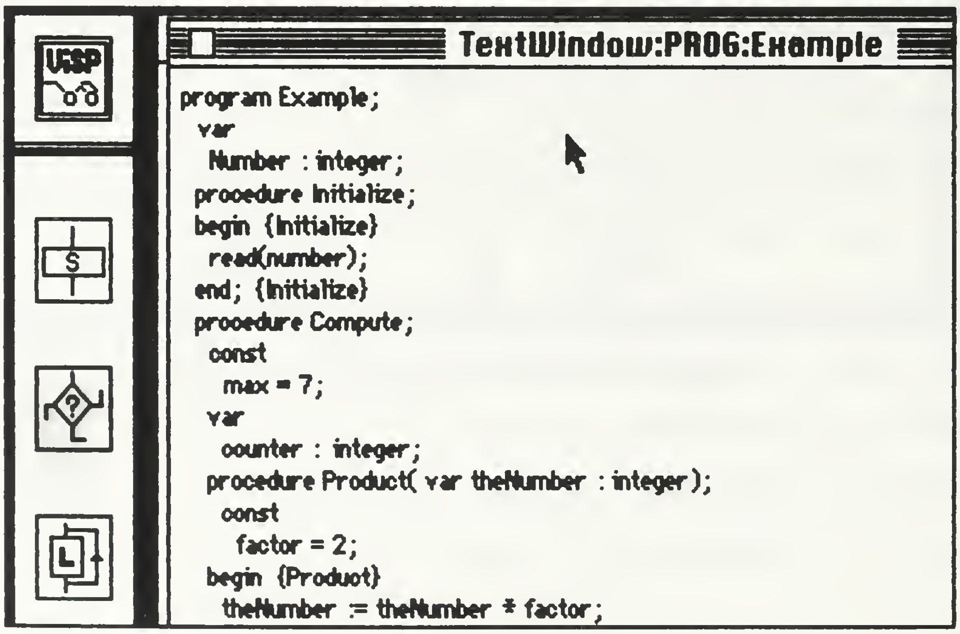


Figure B.10 Printing the Program

```
program Example;
  var
   Number : integer;
  procedure Initialize;
  begin {Initialize}
   read(Number);
  end; {Initialize}
  procedure Compute;
   const
    max = 7;
   var
    counter : integer;
   procedure Product( var TheNumber : integer);
    const
     factor = 2;
   begin {Product}
     TheNumber := TheNumber * factor;
   end; {Product}
  begin {Compute}
   counter := 0;
   while counter < max do
     begin
      Product(Number);
      counter := counter + 1;
     end;
  end; {Compute}
  procedure Output;
  begin {Output}
   write(Number);
  end; {Output}
begin {Example}
  Initialize;
  Compute;
  Output;
end. {Example}
```

Figure B.11 Example.Txt

APPENDIX C

VISP DATA TYPES

```
const

   MaxTypes    = 18;
   MaxWindows = 6;
   MaxLocals   = 6;

type

  {Enumerated types for interface}
  ObjectStateType = (Selected, NotSelected);
  FileStateType     = (Shell, OpenProgram);
  ScreenModeType  = (Select, Connect, Null);

  {Type Table: used for naming Objects and Buildwindows}
  TypeRec = record
    Type_ID    : INTEGER;
    Title        : Str255;
  end;
  TypeTable = array[1..MaxTypes] of TypeRec;

  {Windows Table: used for storing pointers to open windows
                        and handles to their scrollbars }
  WindowsRec = record
    Obj_ID       : INTEGER;
    TheWindow : WindowPtr;
    HScroll      : ControlHandle;
    VScroll      : ControlHandle;
  end;
  WindowsTable = array[1..MaxWindows] of WindowsRec;
```

```
{Locals Record: used to record constants, types and variables
                 associated with a Program or a Procedure}
CRec = record
   ConID : Str255;
   ValID : Str255;
end;
CArray = array[1..MaxLocals] of CRec;

TRec = record
   TypID : Str255;
   DefID : Str255;
end;
TArray = array[1..MaxLocals] of TRec;

VRec = record
   VarID : Str255;
   VtyID : Str255;
end;
VArray = array[1..MaxLocals] of VRec;

LocRec = record
   Consts : CArray;
   Types  : TArray;
   Vars   : VArray;
end;

{Object Record: used to store unique ID number, name, type and location}
ObjPtr = ^ObjRec;
ObjRec = record
   Obj_ID  : INTEGER;
   OType   : INTEGER;
   Name    : Str255;
   Frame   : Rect;
   ObjLink : ObjPtr;
end;
```

```
{Procedure Record: used to store ID number, formal parameters
                   and local constants, types and variables}
ProPtr = ^ProcRec;
ProcRec = record
   Owner_ID          : INTEGER;
   Var1,Var2,Var3    : BOOLEAN;
   ID1,ID2,ID3       : Str255;
   T1,T2,T3          : Str255;
   Locals            : LocRec;
   ProcLink          : ProPtr;
end;

{Basic Record: used to store ID number and contained statements}
BasicPtr = ^BasicRec;
BasicRec = record
   Owner_ID    : INTEGER;
   Statement1  : Str255;
   Statement2  : Str255;
   Statement3  : Str255;
   Statement4  : Str255;
   Statement5  : Str255;
   BasicLink   : BasicPtr;
end;

{Call Record: used to store Call's ID,
              parent Procedure's ID and actual parameters}
CallPtr = ^CallRec;
CallRec = record
   Call_ID    : INTEGER;
   Proc_ID    : INTEGER;
   ActualID1  : Str255;
   ActualID2  : Str255;
   ActualID3  : Str255;
   CallLink   : CallPtr;
end;
```

```
{If Record: used to store If's ID number and Test Expression, as
            well as the ID numbers of siblings Then and Else}
IfPtr = ^IfRec;
IfRec = record
  Owner_ID : INTEGER;
  Then_ID  : INTEGER;
  Else_ID  : INTEGER;
  TestExpr : Str255;
  IfLink   : IfPtr;
end;

{Case Record: used to store Case's ID number, Selector and Case
              Constants, as well as the IDs of siblings}
CasePtr = ^CaseRec;
CaseRec = record
  Owner_ID   : INTEGER;
  Case1_ID   : INTEGER;
  Case2_ID   : INTEGER;
  Case3_ID   : INTEGER;
  Case4_ID   : INTEGER;
  Selector   : Str255;
  CaseConst1 : Str255;
  CaseConst2 : Str255;
  CaseConst3 : Str255;
  CaseConst4 : Str255;
  CaseLink   : CasePtr;
end;

{While Record: used to store While's ID number and Test Expression}
WhilePtr = ^WhileRec;
WhileRec = record
  Owner_ID  : INTEGER;
  TestExpr  : Str255;
  WhileLink : WhilePtr;
end;
```

```
{For Record: used to store For's ID number, Index Variable,
            Initial Value, Final Value and Incrementor}
IncrType = (Up,Down);
ForPtr = ^ForRec;
ForRec = record
   Owner_ID    : INTEGER;
   IndexVar    : Str255;
   IValue      : Str255;
   Incrementer : IncrType;
   FValue      : Str255;
   ForLink     : ForPtr;
end;


{Has Connections Record: used to store the containing
                         Object's ID, the two Objects connected,
                         and the two points of connection}
HCPtr = ^HCRec;
HCRec = record
   Owner_ID : INTEGER;
   ObjOne   : INTEGER;
   ObjTwo   : INTEGER;
   PtOne    : Point;
   PtTwo    : Point;
   HCLink   : HCPtr;
end;

{Has Objects Record: used to store the containing Object's ID,
                     the contained Object's ID}
HOPtr = ^HORec;
HORec = record
   Owner_ID : INTEGER;
   Obj_ID   : INTEGER;
   HOLink   : HOPtr;
end;
```

```
var

  Types       : TypeTable;        {The Type Table}
  Windows     : WindowsTable;     {The Windows Table}
  Globals     : LocRec;           {The Globals Table}

  ObjHead     : ObjPtr;           {The Objects Table}
  ProcHead    : ProPtr;           {The Procedure Table}
  BasicHead   : BasicPtr;         {The Basic Table}
  IfHead      : IfPtr;            {The If Table}
  CaseHead    : CasePtr;          {The Case Table}
  ForHead     : ForPtr;           {The For Table}
  WhileHead   : WhilePtr;         {The While Table}

  CallHead    : CallPtr;          {The Call Table}
  CallPlace   : CallPtr;          {PlaceHolder for Call Table}
  CallKey     : INTEGER;          {Key for Call Table}

  HCHead      : HCPtr;            {The Has Connections Table}
  HCPlace     : HCPtr;            {PlaceHolder for Has Connections Table}
  HCKey       : INTEGER;          {Key for Has Connections Table}

  HOHead      : HOPtr;            {The Has Objects Table}
  HOPlace     : HOPtr;            {PlaceHolder for Has Objects Table}
  HOKey       : INTEGER;          {Key for Has Objects Table}
```

APPENDIX D

OBJECT IMAGES

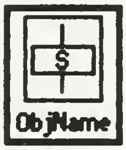


Figure D.1 The Sequencer

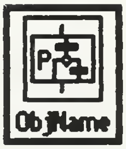


Figure D.2 The Procedure

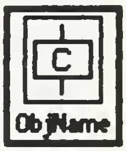


Figure D.3 The Call

Figure D.4 The Basic

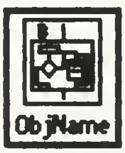


Figure D.5 The Block

Figure D.6 The Branch

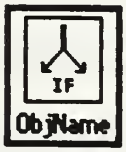


Figure D.7 The If

Figure D.8 The Case

Figure D.9 The Loop

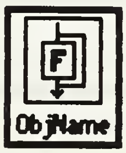


Figure D.10 The For

Figure D.11 The While

APPENDIX E

ALERTS

Figure E.1 Maximum Objects Alert

Figure E.2 Maximum Windows Alert

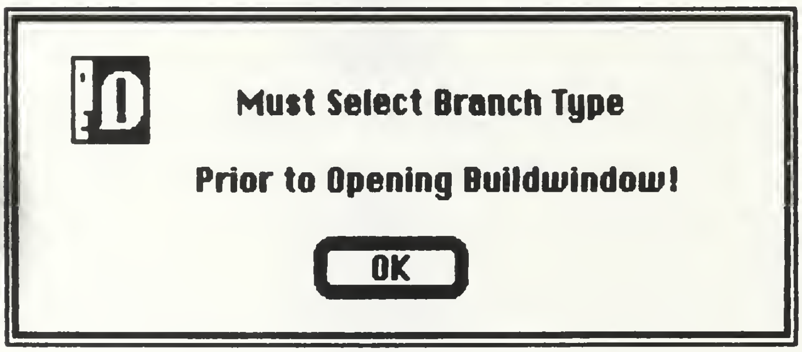


Figure E.3 Select Branch Type Alert

Figure E.4 Fully Connected Alert

Figure E.5 Begin Established Alert

Figure E.6 End Established Alert

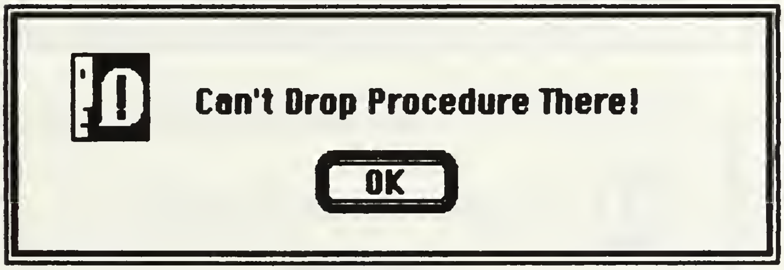


Figure E.7 Cannot Contain Procedure Alert

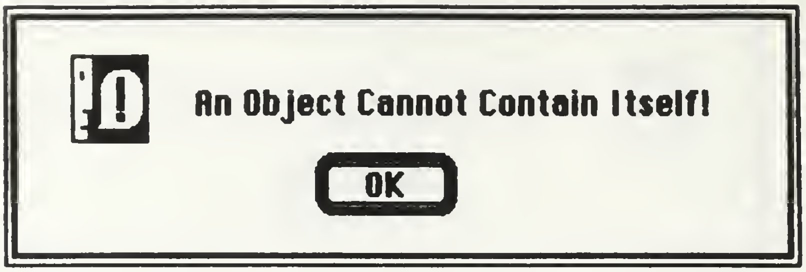


Figure E.8 Cannot Contain Self Alert

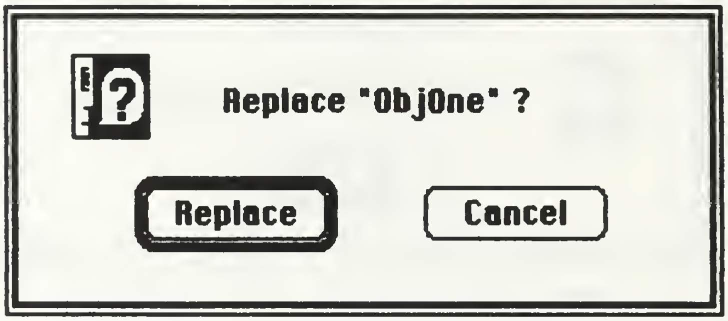


Figure E.9 Replace Object Alert

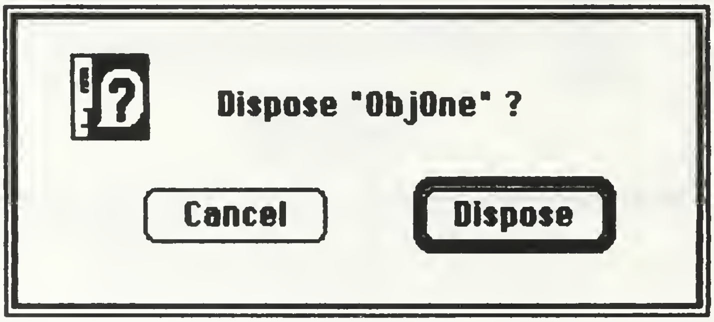


Figure E.10 Dispose Object Alert

Figure E.11 Cannot Connect Procedure Alert

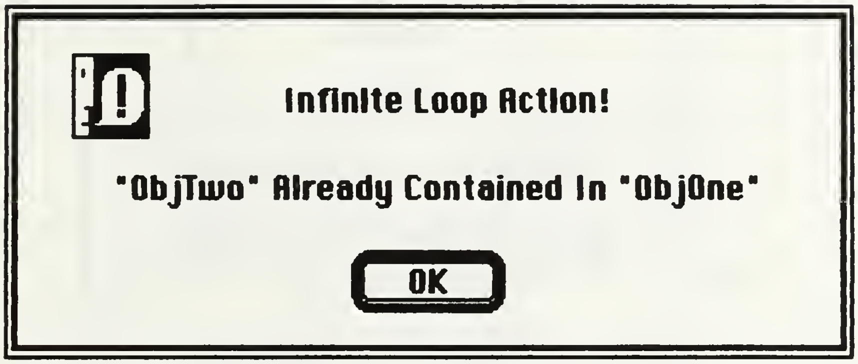


Figure E.12 Infinite Loop Alert

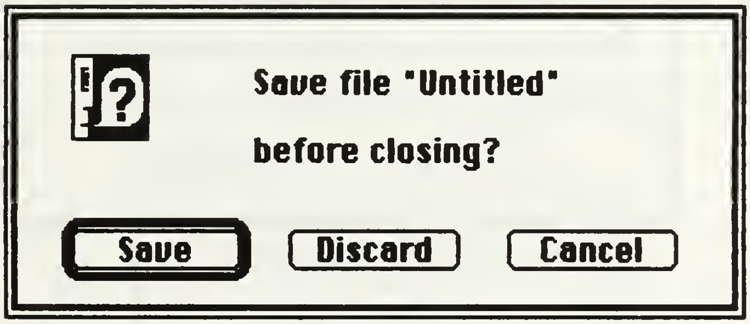


Figure E.13 Save Program Alert

APPENDIX F

DIALOGS

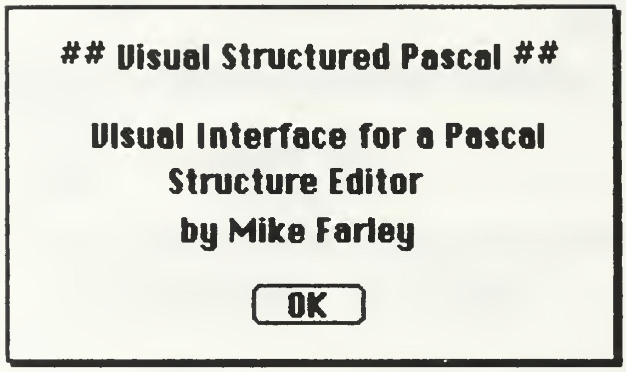


Figure F.1 About ViSP Dialog

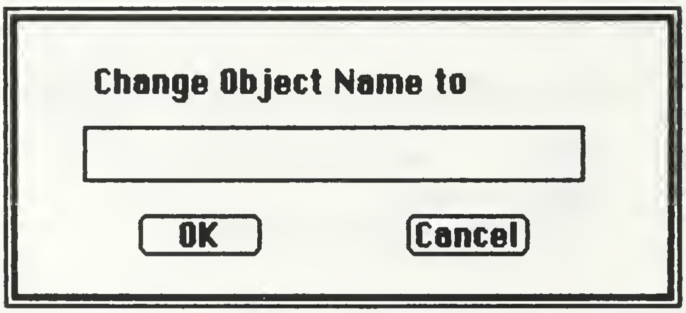


Figure F.2 Change Object Name Dialog

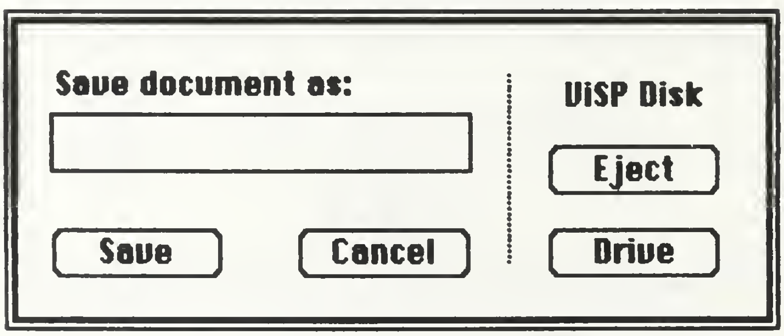


Figure F.3 Save As Dialog

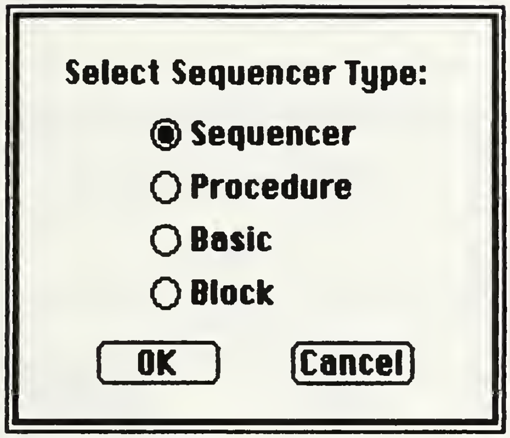


Figure F.4 Select Sequencer Type Dialog

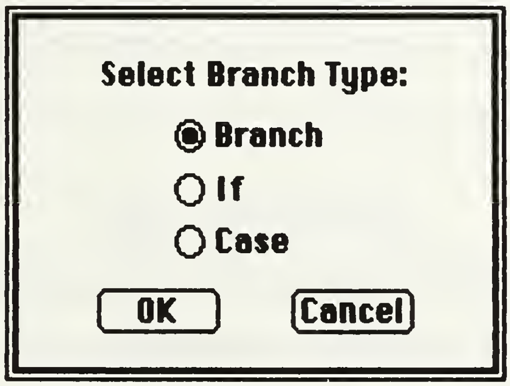


Figure F.5 Select Branch Type Dialog

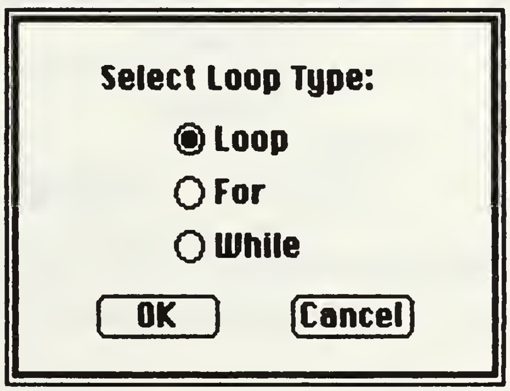


Figure F.6 Select Loop Type Dialog

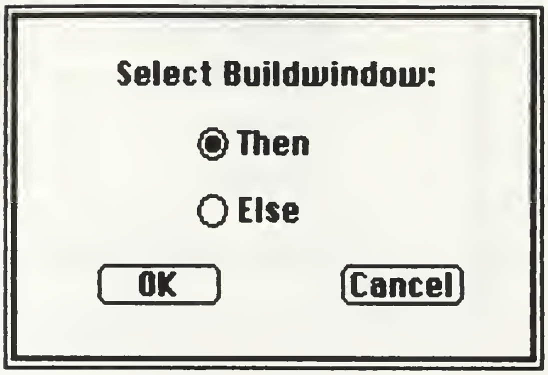


Figure F.7 Select If Buildwindow Dialog

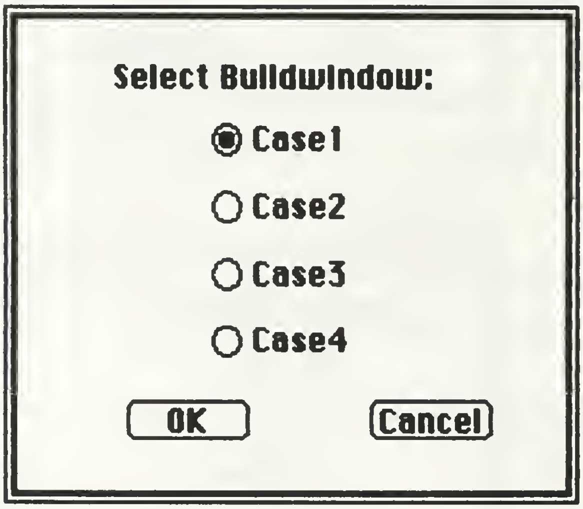


Figure F.8 Select Case Buildwindow Dialog

APPENDIX G

PARAMETER BOXES

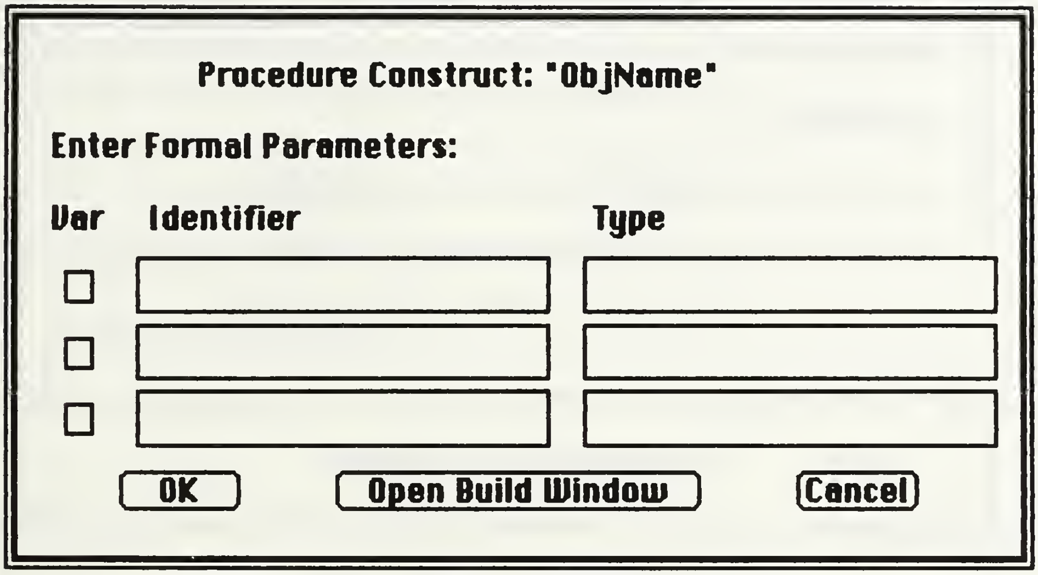


Figure G.1 Procedure Parameter Box

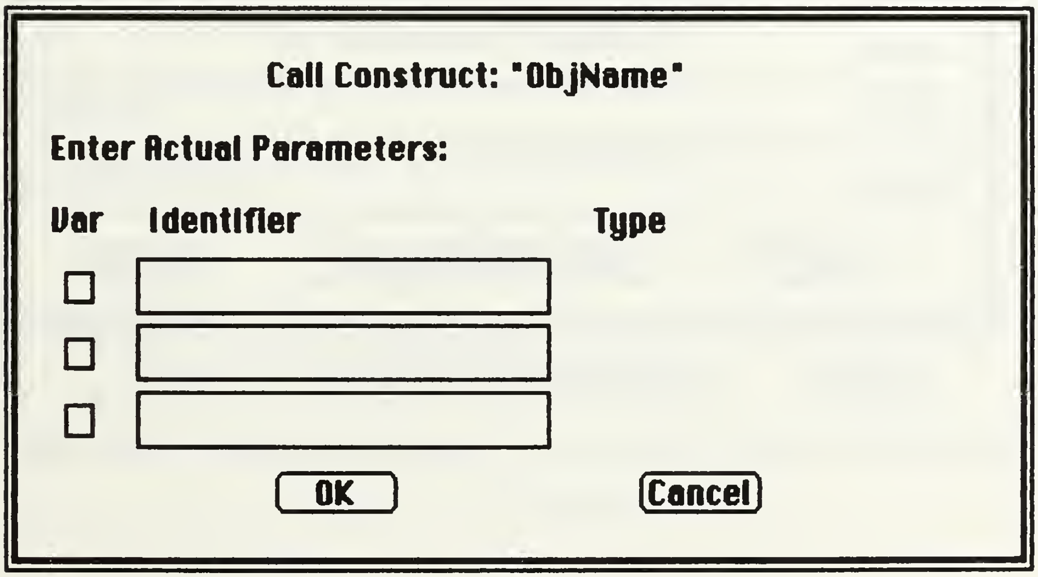


Figure G.2 Call Parameter Box

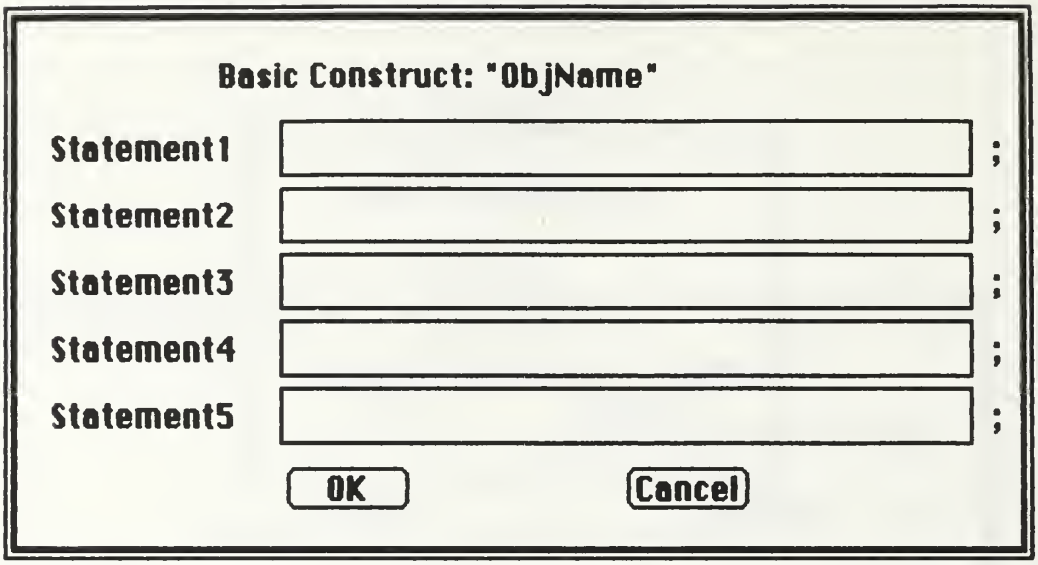


Figure G.3 Basic Parameter Box

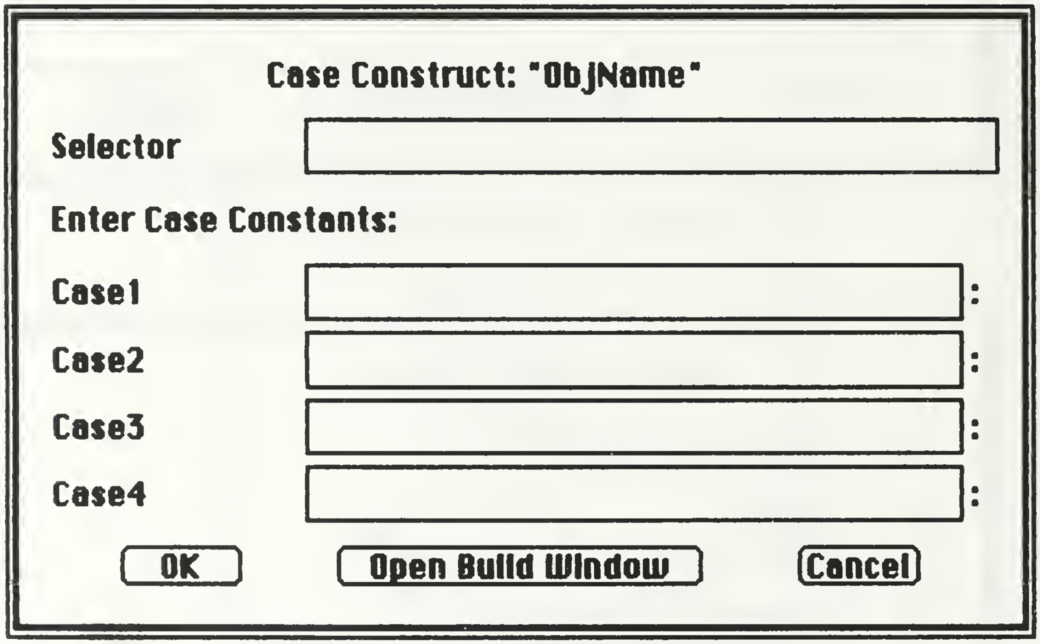


Figure G.4 Case Parameter Box

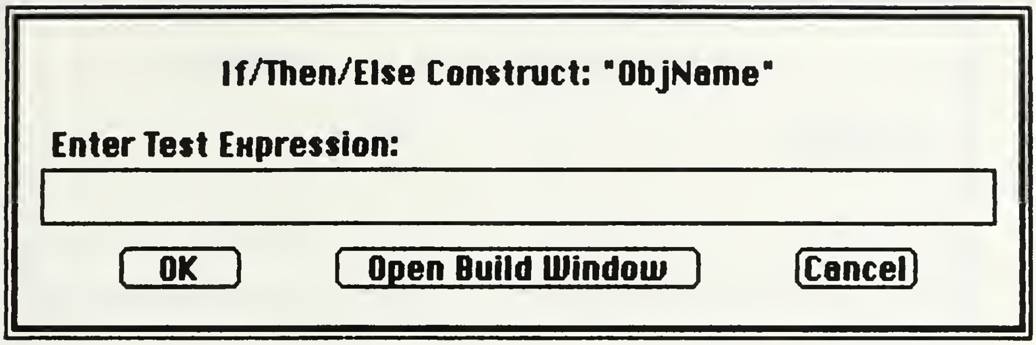


Figure G.5 If Parameter Box

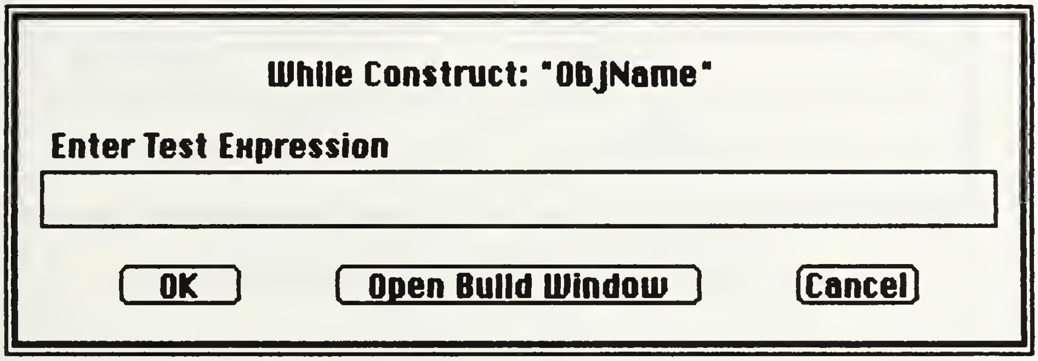


Figure G.6 While Parameter Box

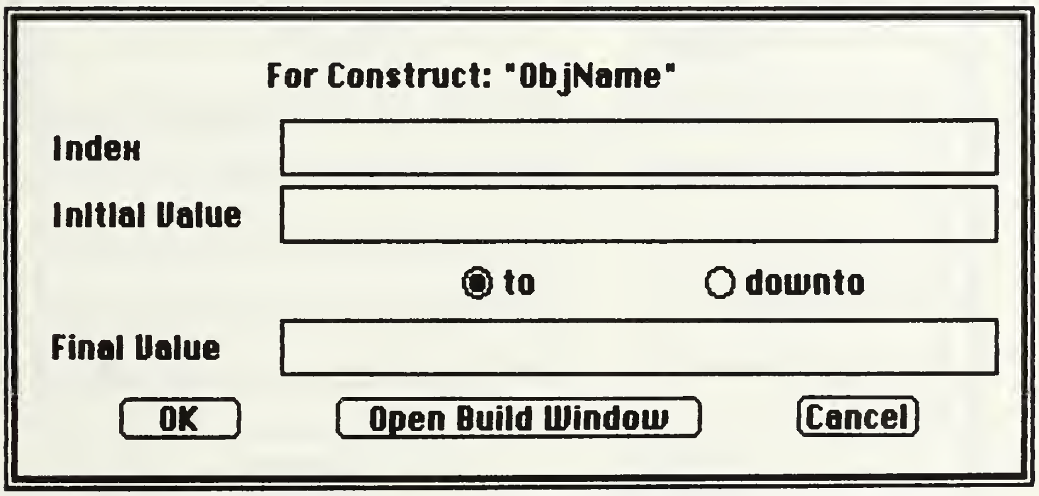


Figure G.7 For Parameter Box

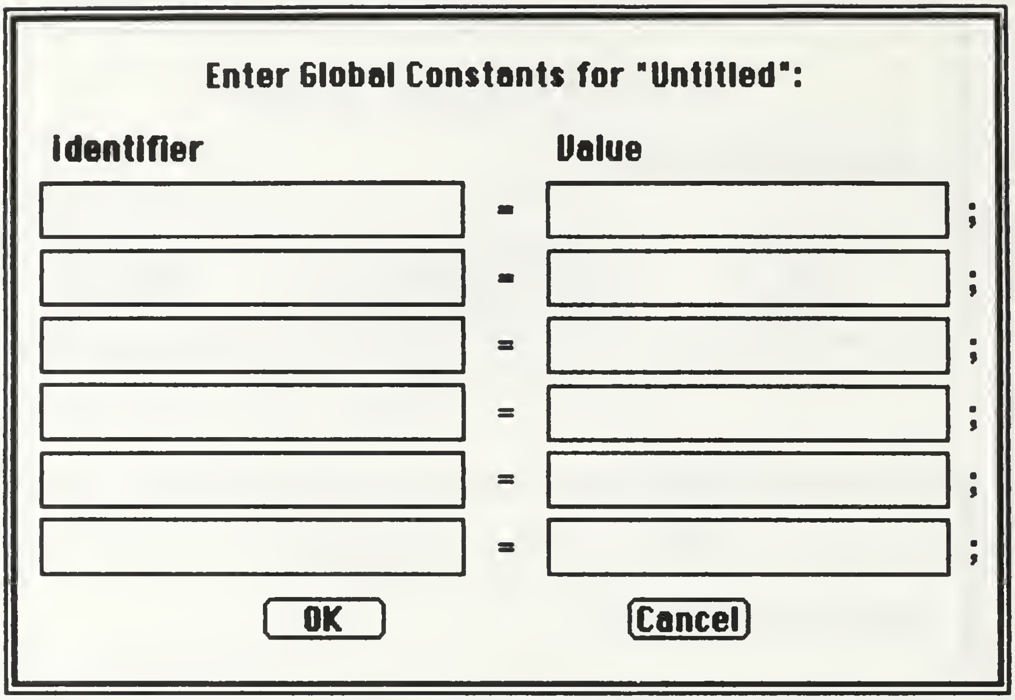


Figure G.8 Constants Parameter Box

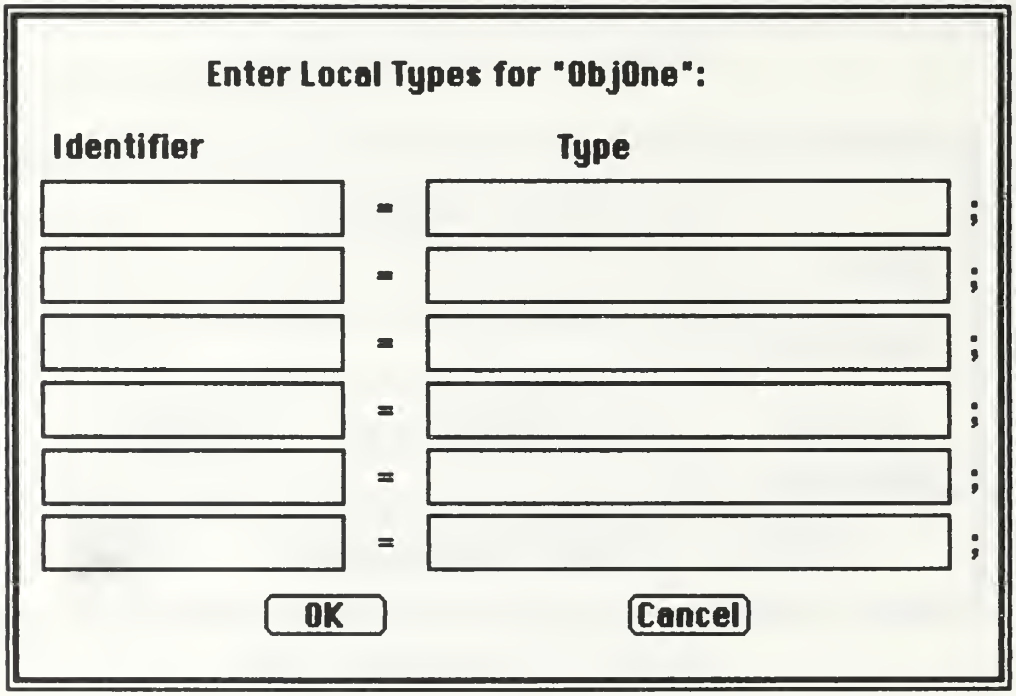


Figure G.9 Types Parameter Box

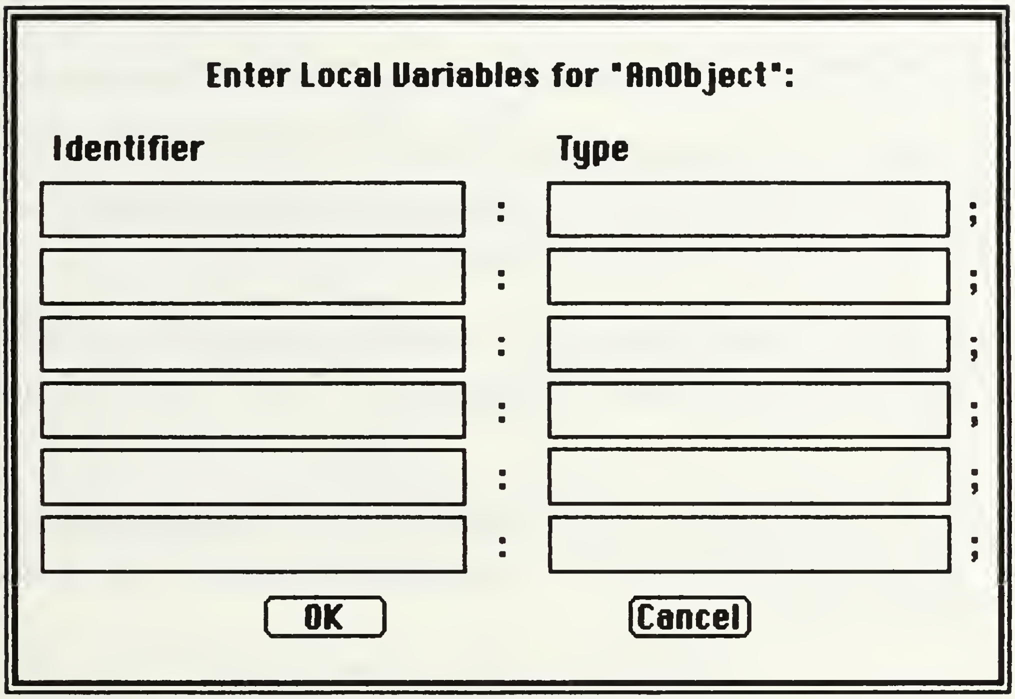


Figure G.10 Variables Parameter Box

LIST OF REFERENCES

1. Hansen, W.R., "User Engineering Principles for Interactive Systems," Interactive Programming Environments, by D. Barstow, E. Sandewall and H. Schrobe, eds., pp. 217-231, McGraw-Hill, 1984.

2. Apple Computer, Inc., Inside Macintosh, Promotional Edition, 1985.

3. Glinert, E.P., "Towards Second Generation Interactive, Graphical Programming Environments," Workshop on Visual Languages, pp. 61-70, IEEE Computer Society Press, June 1986.

4. Glinert, E.P. and Tanimoto, S.L., "PICT: An Interactive, Graphical Programming Environment," Computer, v. 17, pp. 7-25, November 1984.

5. Frei, H.P., Weller, D.L., and Williams, R., "A Graphics-Based Programming-Support System," Computer Graphics, v. 12, pp. 43-49, August 1978.

6. Jacob, R.J.K., "A State Transition Diagram Language for Visual Programming," Computer, v. 18, pp. 51-59, August 1985.

7. Witty, R.W., "Dimensional Flowcharting," Software Practice and Experience, v. 7, pp. 553-584, 1977.

8. Azuma, M., Tabata, T., Oki, Y., and Kamiya, S., "SPD: A Humanized Documentation Technology," IEEE Transactions on Software Engineering, v. 11, pp. 945-953, September 1985.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
4.	Curricular Officer, Code 37 Computer Technology Curricular Office Naval Postgraduate School Monterey, California 93943	1
5.	Associate Professor Daniel L. Davis, Code 52vv Department of Computer Science Naval Postgraduate School Monterey, California 93943	5
6.	Associate Professor Bruce J. MacLennan, Code 52ml Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
7.	LCDR Michael F. Farley Patrol Squadron One FPO San Francisco, California 96601	2